AN LALR (1) PARSER GENERATOR

by

D. K. CHATURVEDI

CSP
1982
M
CHA
LAL

COMPUTER SCIENCE PROGRAMME

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

JULY, 1982

AN LALR (1) PARSER GENERATOR

A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

by

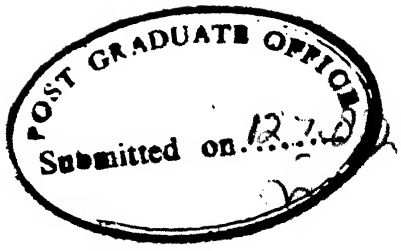
D. K. CHATURVEDI

to the
COMPUTER SCIENCE PROGRAMME
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR
JULY, 1982

5 JUN 1984

CENTRAL LIBRARY
Acc. No. A 82759

CSP-1985-M-CHA--LAI



CERTIFICATE

This is to certify that the thesis entitled "AN LALR(1) PARSER GENERATOR" has been carried out by Sri D.K. CHATURVEDI under my supervision and has not been submitted elsewhere for the award of a degree.

H.V. Sahasrabuddhe

DR. H.V. SAHASRABUDDHE
PROFESSOR
COMPUTER SCIENCE PROGRAM
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

KANPUR
JULY 1982

ACKNOWLEDGEMENT

I am grateful to Dr. H.V. Sahasrabuddhe, my thesis supervisor, for his constant help and guidance throughout this project.

I also thank my friends and colleagues especially Capt. A.V. Subramanian, Ajay Tyagi and G.S. Kumar for having made my stay at I.I.T. Kanpur pleasant.

Finally, I thank Mr. M.C. Gupta for his excellent typing.

ABSTRACT

The Thesis "An LALR(1) PARSER GENERATOR" describes the design and implementation of an LALR(1) parser generator system. Automatic parser generators are an essential tool in Translator Writing Systems. By using this program, a Programmer can develop a parser for very large grammers in a few days. LALR(1) has been chosen because the tables obtained by it are considerably smaller than LR(1) tables, yet most common syntactic constructs can be conveniently expressed by an LALR(1) grammer. Besides, ambiguous grammers augmented with precedence and associativity declarations can also be specified. The system provides facilities to the programmer to incorporate error recovery routines in his parser. The system has been tested on several grammers, including the PASCAL grammer.

CONTENTS

CHAPTER	TITLE	PAGE
1.	INTRODUCTION	1
2.	LALR(1) ALGORITHM	3
3.	IMPLEMENTATION OF THE LALR(1) ALGORITHM	10
4.	INSTRUCTIONS TO THE USER	22
5.	CONCLUSIONS	30
	APPENDIX	31
	REFERENCES	33
	PROGRAM LISTINGS	

CHAPTER I

INTRODUCTION

Automatic Parser generatorsare essential tools for compiler writers. Syntax analysis is the best understood aspect of compilers and many good algorithms have been developed to generate parsers automatically. Most of the generators besides producing the parser, help the user to write input grammers correctly. The generators check the input grammer for ambiguity and give suitable diagnostic messages in case of errors in the input grammer. Indeed it is practically impossible to write a parser by hand for some of the popular parsing methods such as LR. By using such tools the programmer can develop a Parser in a few days for very large grammers.

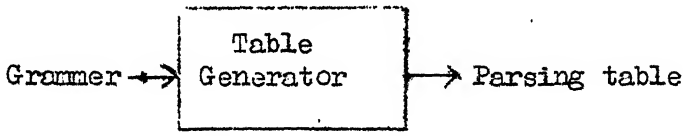
Two of the widely used Parsing techniques are LL(1) and LR(1), also known as TOP-DOWN and BOTTOM-UP Parsing. The reader is referred to chapters 5,6 of [1] for a good treatment for both of the above techniques. Of the above two methods, LR is more powerful as it accepts a larger class of grammers.

The first paper on LR Parsers was by KNUTH in 1965. However direct implementation of his method was very inefficient. Later more practical methods which are variants of the LR method, such as LALR(1), SLR(1) and LR(0) were developed. These consume less space and are efficient to implement, however they do not accept all the grammers accepted by LR(1).

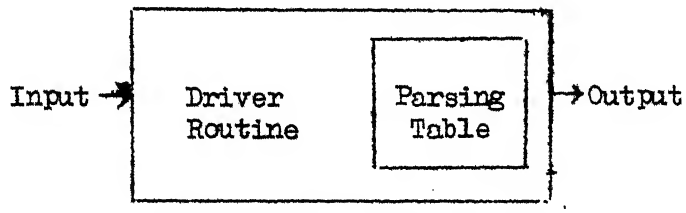
We chose the LALR(1) technique for our implementation. This method has been chosen because the tables obtained by it are considerably smaller than LR tables, yet most common syntactic constructs can be conveniently expressed by an LALR(1) grammer. Besides, just because a

technique rejects a particular grammer does not necessarily mean that it will reject all Grammers for the same language. The largest practically recognizable class is called the deterministic languages, and it is precisely the class defined by the LR(K) grammers for any $K \geq 1$, or by LALR(1) grammers, or by SLR(1) grammers. Thus, we do not sacrifice any languages when we restrict to an LALR(1) grammer.

An LALR(1) Parser generator system consists of two parts, a driver routine and a parsing table. The driver routine is the same for all LALR(1) Parsers; only the parsing table changes from one parser to another. Thus generating the parser requires generating the parsing table from the input grammer and then integrating the parsing table with the driver routine.



(a) Generating the Parser



(b) Operation of the Parser

Many algorithms for LALR(1) Parser construction have appeared in the literature. The algorithm used here is due to B.B. Kristensen and O.L. Madsen [5]. Chapter 2 of the thesis describes the algorithm, chapter 3 discusses the implementation and chapter 4 describes how the system is to be used.

The following grammers are used for examples in the rest of the chapters:

Grammer G1: E = E + E | E * E | ID

Grammer G2: E = E + T | E * T | T
T = ID

CHAPTER 2

LALR(1) ALGORITHM

This chapter describes the algorithm used in the implementation. It is assumed that the reader is familiar with the terminology and conventions concerning LR Grammers and LR Parsing Theory. The reader is referred to chapter 6 of [1] for a good treatment of LR Parsers. The algorithm used in our implementation is based on [5] . This algorithm was chosen because it directly results from the definition of LALR(1) and is easy to understand and implement besides being efficient. Some of the definitions used are given here.

A context-free grammer is assumed to have the form $G = (N, \Sigma, P, S)$, where 'N' is a finite set of Non-Terminals, 'Σ' is a finite set of terminals, 'P' is a finite set of Productions and 'S' is the start symbol. We assume that the grammer is reduced and free of useless symbols. We extend the grammer by adding the production $S' \rightarrow S$. The augmented grammer is denoted as G'.

The following convention is used:

Symbols such as	α, β, γ	are in $(N \cup \Sigma)^*$
" "	a, b, c	are in Σ
" "	v, x, y	are in Σ^*
" "	A, B, C	are in N
" "	X, Y, Z	are in $(N \cup \Sigma)$

The Null string is denoted by e or NUL.

Definition 2.1:-

An LR(0) item of a Grammer G is a production with a dot at some position of the right side. It is denoted as a 2-tuple $<p,j>$ where 'p' is the production number and 'j' is the position of the dot.

Thus $A \rightarrow X.YZ$ is an item.

Intutively an item is a partially recognized production. For e.g. the above item tells us that we have already recognized X in the input string and expect to see a string derivable from 'YZ'.

Definition 2.2:-

A valid item for a viable prefix $\gamma\alpha$ is an item $A \rightarrow \alpha.\beta$ such that there is a derivation

$$S' \underset{rm}{\overset{*}{\Longrightarrow}} \gamma A w \underset{rm}{\Longrightarrow} \gamma\alpha\beta w.$$

These valid items can be pre-computed for a given grammer G. In fact the LR(0) sets of items is precisely the set of valid items. The reader is referred to chapter 6 of [1] for the algorithm on LR(0) items construction.

Definition 2.3:-

A State is a set of items. The closure of a state I is defined as follows:

1. Every item in I is in closure (I).
2. If $A \rightarrow \alpha . B\beta$ is in closure(I) and
   $B \rightarrow \gamma$ is a production, then add
   $B \rightarrow .\gamma$ to I, if it is not already there.

Definition 2.4:-

GOTO(I,X), is defined to be the closure of the set of all items $A \rightarrow \alpha X.\beta$ such that $A \rightarrow \alpha.X\beta$ is in state I. This definition can be extended to GOTO(I, α).

Definition 2.5:-

Kernel of a state S is defined to be the set of items $A \rightarrow \alpha . \beta$ in the state such that $|\alpha| > 0$. The only exception to this rule is the start state where $S' \rightarrow .S$ is in the Kernel.

We can compute the LALR(1) lookaheads by using the Kernel items alone. This results in large saving in storage space needed for storing the items in the states.

Definition 2.6:-

M_o is used for LR(0) sets of items. The algorithm to compute M_o is given in chapter 3.

Definition 2.7:-

PRED is defined as follows:

Let $T \in M_o$, $X \in (N \cup \Sigma)$ and $\alpha \in (N \cup \Sigma)^*$ then

$$PRED(T,\alpha) = \begin{cases} \{T\} & \text{if } \alpha = e \\ \bigcup \{ PRED(S,\alpha') \mid GOTO(S,X)=T \} & \text{if } \alpha = \alpha'X \end{cases}$$

Description of the LALR(1) Algorithm:-

We need to compute the LALR(1) lookaheads only for those items which are of the type $[A \rightarrow \alpha .]$. Informally the LALR(1) lookaheads of an item $[A \rightarrow \alpha .]$ in a state T may be described as the set of terminals that may appear on input, if during parsing, the reduction $A \rightarrow \alpha$ can be applied in state T. We are thus interested in knowing all states where parsing may be resumed after the reduction by $A \rightarrow \alpha$ is performed in T.

Let S be a state containing the item "$B_i \rightarrow \varphi_i \ .A \ \delta_i$" . Then by closure operation it will also contain the item $A \rightarrow . \ \alpha$. Let $GOTO(S, \alpha) = T$. Then we see that Parsing can be resumed in state S after the reduction $A \rightarrow \alpha.$ in T and will continue in state R where $R = GOTO(S,A)$. $PRED(T,\alpha)$ is exactly the set of such states where parsing can be resumed after the reduction $A \rightarrow \alpha.$ in state T.

State R will contain items of the form $B_i \rightarrow \varphi_i \ A. \ \delta_i$ for $i=1,2...p, p > 0$. Since to compute LALR(1) lookaheads we are interested in knowing all terminals that can appear on input after the reduction, we have to find such R states for each of the states in $PRED(T, \alpha)$.

The terminal symbols that can be read in state R are

$$\bigcup \{First(\delta_i) \quad i = 1,2,....p\}$$

If some $\delta_i \overset{*}{\Rightarrow} e$ then we have to reduce by "$B_i \rightarrow \varphi_i \ A \ \delta_i$". Then we have to find out the terminal strings that may follow after this reduction i.e. we have to compute the LALR(1) lookahead set for $[B_i \rightarrow \varphi_i.A \ \ \delta_i, S \ ]$. Thus we get a recursive LALR(1) algorithm.

The above description was an informal description of the LALR(1) algorithm. We give below the formal mathematical definition of the LALR(1) and describe the algorithm in a Pascal-like language.

Definition 2.8:-

If $T \in M_0$ then

$$TRANS(T) = \bigcup \{First_1(\beta) \mid A \rightarrow \alpha . \ \beta \in Kernel(T)\} - \{e\}$$

Definition 2.9:-

If $A \in N$ and $S \in M_o$ then

$$LA(A,S) = TRANS(GOTO(S,A)) \cup (\{ LALR(1)([B \rightarrow \varphi.A\delta], S) \mid [B \rightarrow \varphi.A\delta] \in S \wedge \delta \overset{*}{\Rightarrow} e \})$$

From the above definitions and the definition of LALR(1) lookaheads we can write:-

For all items $[A \rightarrow \alpha . \beta]$ and $S,T \in M_o$

$$LALR(1)([A \rightarrow \alpha.\beta], T) = \cup\{ LA(A,S) \mid S \in PRED(T, \alpha) \}$$

The algorithm is described in a Program like notation on next pag Since M_o is in general full of cycles, hence we must keep track of items for which LALR(1) set has already been computed, so that the algorithm is efficient and recursion is guranteed to terminate. This is done by the variable 'DONE' which stores all the (non-terminal X state) pairs for which LALR procedure has already been called.

Example:-

Fig. 2.1 gives the LR(0) machine for Grammer G1 given in Chapter 1.

Fig. 2.2 shows the prodecessor tree that will be traversed if a call LALR(1)([E → ID.] ,2) is made. A box contains the number of the state and the item considered and the lookahead added in that state. The interior nodes correspond to states in which a recursive call is made. The leaves correspond to states in which recursion is stopped.

```
Procedure   LALR1(I:ITEM;T:STATE;var LA: lookaheadset);
  var       DONE:array[1..DONEMAX] of record
                          NONTERMINAL,STATE:integer
                                          end;

  Procedure   TRANS(T:STATE);
    begin     for all items [A →α.β] do
              LA:=LA+FIRST(β)-{e}
              end for
    end Trams;
  Procedure   LALR(I:ITEM;T:STATE);
    begin     assume I=[A → α.β];
        for  S ∈ PRED(T,α) where (A,S)∉DONE do
             DONE:=DONE+(A,S);(* add to DONE array *)
             TRANS(GOTO(S,A));
          for  [B → φ.Aδ]∈ S∧ δ⇒* e  do
            if  LALR has been computed
                      then  LA:=LA+LASET([B → φ.Aδ],S)
            else      LALR([B → φ.Aδ],S)
          end for
        end for
    end LALR;
  begin
        DONE:=0;LA:=[ ]; assume I=[A →α.β];
        if A=S' then LA:= {$} else LALR(I,T)
  end LALR1;
```

NOTE: assume has no action associated with it.
$ denotes the end of Input and is not a symbol in $(N \cup \Sigma)$
LASET is the lookahead symbols computed for that item.

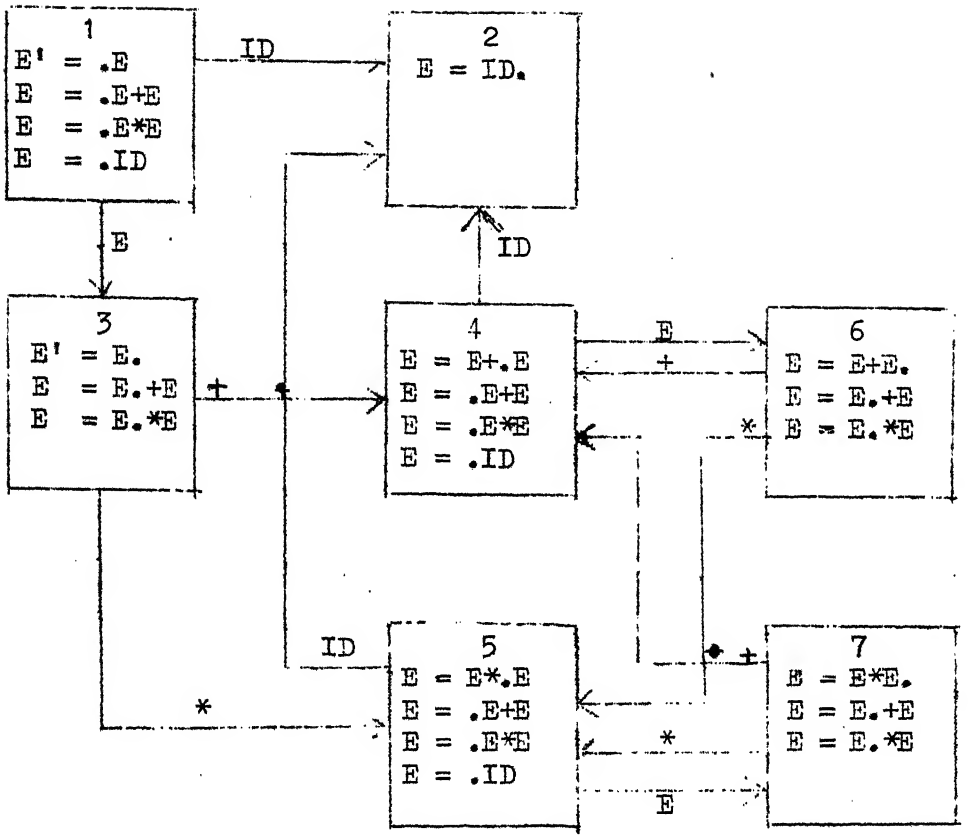
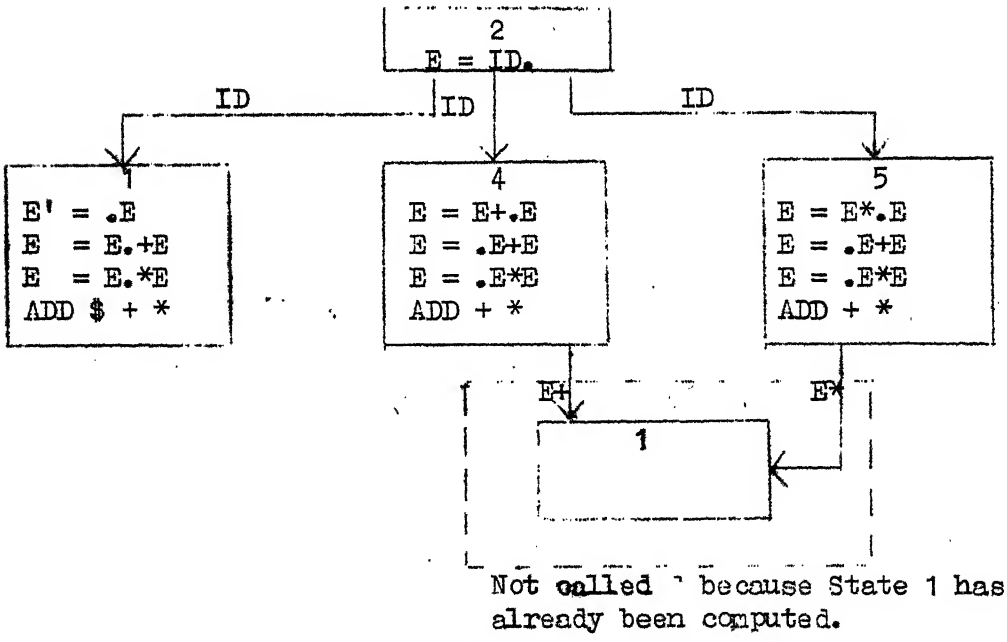


FIGURE 2.1



Not called because State 1 has already been computed.

FIGURE 2.2

CHAPTER 3

IMPLEMENTATION OF THE LALR(1) ALGORITHM

This chapter gives the implementation details of the algorithm. Most of the important data structures and procedures are described in this chapter.

NT is used as an abbreviation for Non-Terminal.
T is used as an abbreviation for Terminals.

3.1 DATA STRUCTURES:-

(a) Arrays TSAVE and NTSAVE are used to store the names of the identifiers declared as Terminals and Non-Terminals respectively.

NUL, $, ERROR are pre-declared Terminals.
STARTP is a pre-declared Non-Terminal.
$ is a pre-declared symbol used to denote end of input file.

(b) To store the productions, four arrays NTARRAY, NTSONS, PRODARRAY and PROD are used.

Since one NT can be L.H.S. of many productions, hence NTARRAY points to the first production of each NT into the PRODARRAY. The rest of the productions are linked in PRODARRAY. NTSONS keeps counts of the number of productions, for which a particular NT is the L.H.S. of the production.

PRODARRAY stores the description of all productions. Each production is described as a record whose fields are explained

below:

```
Precedence: integer; (* Precedence of the Production *)
NT        : integer; (* L.H.S. of the Production *)
NEXTPROD  : integer; (* Points to the next Production for
                        same NT *)
PRODPTR   : integer; (* Pointer to the PROD array where the
                        R.H.S. of the production is stored *)
LENGTH    : integer; (* no. of symbols in the R.H.S. of the
                        production *)
```

The array PROD stores the R.H.S. of the productions. Each element of the array PROD is a record with the following fields:

```
ALPHA : Alphastype ; (* Terminal or NT *)
NUM   : Integer ; (* The number assigned to Alpha *)
```

(c) LAMDA is an array which stores whether a particular non-terminal generates the Null string or not. The value is TRUE if Null string is generated, otherwise it is FALSE

(d) FT is an array which stores the first symbols for each non-terminal.

(e) Each ITEM description is stored in the array ITEMS. An item is of the form $\langle p,j \rangle$ where p is the production number and j is the place of dot on the R.H.S. of the production. ITEMS is an array of recordswhose fields are described below:

```
LAM: Boolean; (* If Production is of the form
                 A -> α.β and β =*=> λ then
                 LAM=TRUE else LAM=FALSE *)
```

```
LADONE: Boolean; (* Is Initially set to false.
                    When LALR Set has been computed for
                    this item then it is set to TRUE *)

PRODNO: integer; (* the number of the production *)

PLACE:  integer; (* the Place of the dot *)

PREDPTR: PREPTR; (* Pointer to linked list of
                    Predecessor states for this item.
                    See definition of Predecessor states
                    in previous chapter *)

LALR: LASET;     (* The lookahead set computed for
                    this item is stored here *)
```

(f) Each State is stored in the array STATES. A State is a collection of ITEMS. We store only the Kernel of each State. Besides the Nul productionswhich are added due to the closure of the State are also stored in the Kernel. Since NUL productions don't have any symbols on the R.H.S. hence we store only the production numbers. All other productions in the Kernel Don't have a dot at position Zero (except the initial State). The description of these items is stored in the ITEMS array. Thus each state is described as a record whose fields are described below:-

```
NOOFITEMS: integer; (* no. of items in the State *)
KPTR:      integer; (* Points to ITEMS array where the
                       ITEMS of the State are described
NPTR:      integer; (* used by search. See description
                       of procedure NEXTSTATE *)
PGOTO: GOTOTYPE;    (* Pointer to a linked list which
                       describes what action has to be
                       taken for each input symbol *)
NULPROD: NULPTR;    (* Pointer to a linked list of
                       NUL Productions in this State *)
```

(g) The GOTO on NT'S is stored in a separate table. This table is very sparse and many states don't have any GOTO entries on NT'S. Hence we have for each NON-Terminal, a list of pairs of the form (Current-State, Nextstate). Thus TGOTO array stores the pointer to a linked list of records whose field descriptions are given below:

```
CURRENTSTATE, NEXTSTATE: integer (* described above *)
PTR: BGOTO; (* Pointer to next element *)
```

(h) TPRED stores the precedence and association described for each terminal. (See Section 4.5). The default value for Precedence is Zero and Associativity is undefined.

(i) TEMP1, TEMP2 and TPARSE arrays are used for outputting the Parse Table. Since many rows of the Parse table are common we eliminate these rows to reduce the table size. All states having common entries point to the same row in the Parse Table.

To look for the common rows, we first see if the number of entries in the rows are same (called length of the row). Since many rows can have same number of entries, hence TEMP1 has a Pointer to entry in TEMP2 for a particular length of the row. In TEMP2 all rows whose lengths are same are linked. Besides they point to the appropriate row in the TPARSE table. See figure 3.1.

3.2 Description of Procedures:-

3.2.1: READINPUT is the first main Procedure which reads the Grammer from the Input file. The Syntax of the Grammer is given in APPENDIX. This Procedure has been written using the Recursive descent technique which is familiar to all students of compilers. The method of error

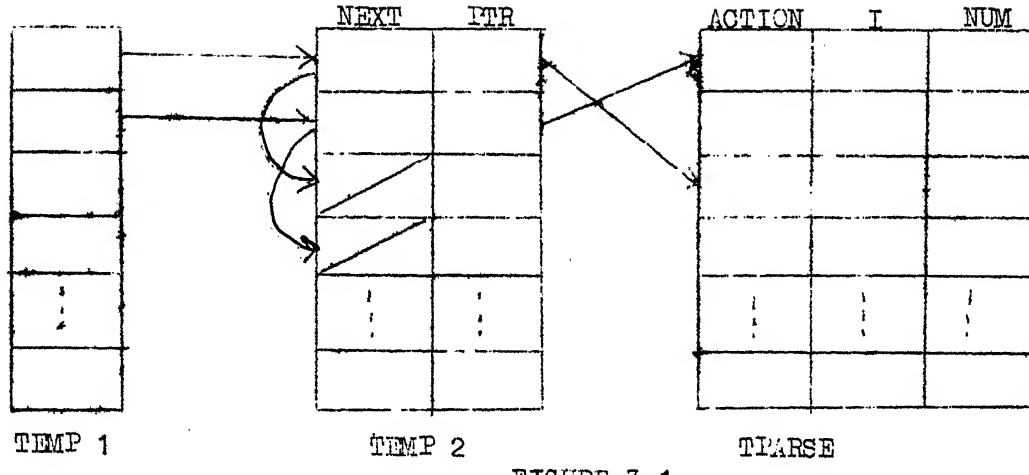


FIGURE 3.1

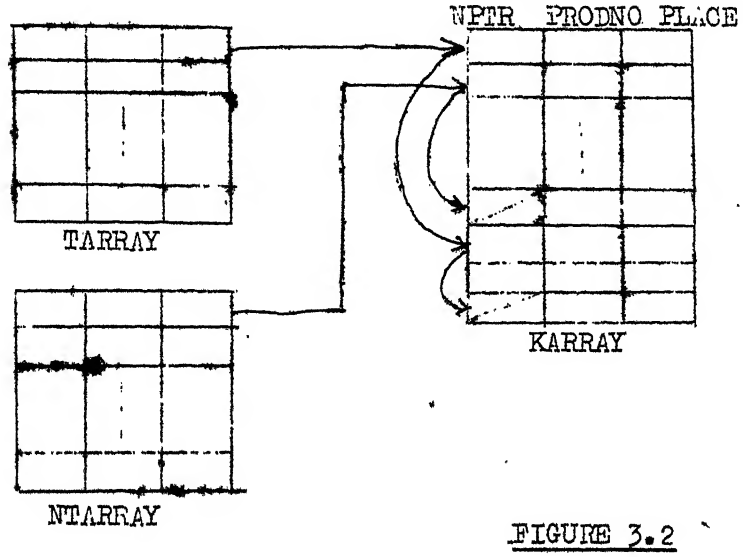


FIGURE 3.2

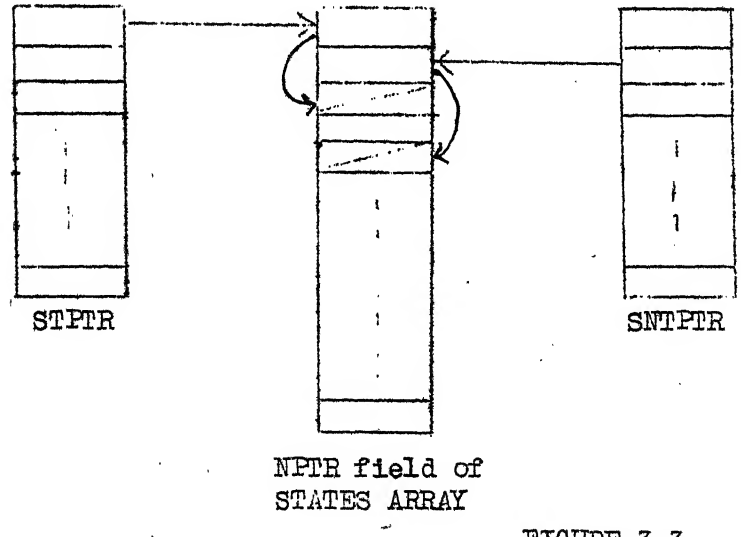


FIGURE 3.3

recovery used is Panic mode. If errors in Input Grammar are detected, the Program outputs the error messages on the OUTPUT file and aborts the Programs after giving the 'PROGRAM ABORTED' message on the TTY. No further description is necessary as the program uses the standard recursive descent technique.

3.2.2: Procedure LAMBDA:

This Procedure computes which Non-Terminals can generate the empty string and which can not. It sets the corresponding entries in the LAMBDA array to TRUE or FALSE depending on whether the NT can generate a NULL string or not. Refer to page 64 of [2] for the algorithm.

3.3.3: Procedure FIRST:

This Procedure computes the First symbols of all Non-Terminals. It updates the array FT with the computed first symbols for each Non-Terminal. Refer to page 65 of [2] for the algorithm.

3.3.4: Procedure PROCITEMS:

This Procedure computes the LR(0) sets of items. Besides all GOTO entries are filled in during this time. In this section STATES denotes the array which stores the states. The algorithm is given below:-

```
begin
STATES: = { CLOSURE( {S' →.S} ) } ;
    repeat
      for  each set of items in STATES and each grammer
           Symbol X such that GOTO(I,X) is not empty and is
           not in STATES
           do  add GOTO(I,X) to STATES
    Until  no more sets of items can be added to STATES
end.
```

Procedure CLOSURE computes the closure of the state. Actually it does more than that. It computes the GOTO(I,X) for all items of the type "$A \rightarrow \alpha.X\beta$". Since GOTO(I,X) has to be searched in the STATES array, the items in GOTO(I,X) are kept in a sorted form. This enables us to compare two states directly, if the number of items in them are same.

KARRAY stores the items of GOTO(I,X). All the items of GOTO(I,X) are linked by the NPTR field of KARRAY. The number of items in GOTO(I,X), the pointer to the first item of GOTO(I,X) and number of X, are stored in TARRAY if X is a terminal otherwise they are stored in NTARRAY. See fig. 3.2 for the data structure representation.

The Procedure NEXTSTATE searches GOTO(I,X) in the STATES array. If GOTO(I,X) is not found then procedure SEARCH adds GOTO(I,X) to states. Procedure NEXTSTATE updates the action table of the current state for both terminals and Non-Terminals. Besides null productions in the current state are added to the Kernel of the current state.

Procedure SEARCH returns the state number if GOTO(I,X) is found, otherwise it adds GOTO(I,X) to states and returns the state number of the new state. Since number of states can be very large, to search fast we link all states whose items have the form "$A \rightarrow \alpha X. \beta$". The Pointer to the first item of the list is kept in STPTR if X is a terminal, otherwise it is kept in array SNTPTR. The states are linked by the NPTR field in the STATES array. See figure 3.3.

3.3.5 Procedure ITEMINTIALISE:-

This procedure initialises all the items in the ITEMS array. Except for the first state, it initialises LADONE to false

since LALR(1) set is not known for the item, PREDPTR to NIL since predecessor states are initially NIL and LAM is set to TRUE if the item is of the form $A \rightarrow \alpha.\beta$ and $\beta \Rightarrow^{*}$ NUL, otherwise it is set to FALSE.

3.3.6 Procedure LALRSET:-

This Procedure has been implemented as described in the previous chapter and is easy to follow.

3.3.7 PRED COMPUTATION:-

We observe that the Procedure LALR is called only when an item is complete in a state or when the item is of the form $B \rightarrow \varphi.A\delta$ and $\delta \Rightarrow^{*} e$. Thus the computation of PRED states is needed for these items only. PRED has been recursively defined earlier. However we use a different non-recursive method to compute PRED states.

The array TGOTO has a linked list of pairs of the form (currentstate, nextstate) for each NT. Thus for a non-terminal A, the linked list will have all the states (in the currentstate field) where as items of the type $B \rightarrow \varphi .A\delta$ appear. The algorithm is described on the next page.

The states which have items of the form $A \rightarrow \varphi B.\delta$ and $\delta \Rightarrow^{*} e$ are initialized by the Procedure ITEMINTIALISE.

Procedure COMPUTE-PRED:-

```
begin
    For all Non-Terminals do
        (* assume A, is the Non-Terminal *)
      For all states in currentstate field of the TGOTO
                                        linked list for A do
          (* assume S is the State *)
        For all Production of the type [A → .α ]in S do
            go forward over symbols of α using GOTO
            Tables and store currentstate in PRED
            field if the state has an item of the
            form A → α.  or  A → φ.B δ and δ ⇒* e
        end for
      end for
    end for
end;
```

3.3.8 Procedure OUTPARSETABLE:-

This Procedure outputs the Parse Table in the file named PARSET. At this stage the shift actions for all states have already been computed and the LALR lookahead sets for all completed items in the states have been computed. This procedure does the following for all states:

1. COPIES the shift entries in the array named PARRAY.
2. Enters the reduce entries in the state for all completed items of the state. These reduce entries are entered in PARRAY.

While trying to enter a Reduce entry for a Terminal the Parser may encounter a Shift-Reduce or Reduce-Reduce conflict. A

Shift-Reduce conflict is resolved as follows:-

a. If Precedence of Terminal involved is greater than the Precedence of the production, then the ACTION is SHIFT and vice-versa.

b. If the Precedence of Terminal and Production are equal then
if the associativity of the Terminal is LEFT then the ACTION IS REDUCE
if the associativity of the Terminal is RIGHT then the action is SHIFT
if the associativity is not defined then error is reported in the OUTPUT file, and action taken is SHIFT

A Reduce-Reduce conflict between Productions P1 and P2 is resolved as follows:-

a. If Precedence of P1 > P2 then reduce by P1 and vice-versa
b. If Precedences are equal then error is reported in the OUTPUT file and action taken is REDUCE by P1.

3. If the state has a shift on 'ERROR' Terminal then all lookahead are enumerated, otherwise the Parser enumerates lookaheads only to the extent necessary to resolve the Parsing action conflicts. Reduce action is indicated for all remaining Terminal Symbols by a production which has the maximum lookahead symbols. This reduces the size of the Parse Table considerably.

4. The current action Row of the state is compared by the Procedure COMPARE with all the ROWS which had the same number of entries. If it is identical with some other ROW then it is not outputted, otherwise it is outputted on the file PARSET.

3.3.9 PROGRAM DRIVER:-

The listing of this Program is given in the end. The parse tables have been generated as an INITPROCEDURE by the previous program. These tables are copied after the declaration section of the program DRIVER is over. The lexical analyzer and error recovery routines have to be written by the user (see chapter 4).

The Program DRIVER consists of two procedures GETACTION and PARSERUN.

The Procedure GETACTION gets the next action given the current-state. It examines all the terminal entries in the row of the current-state. If some terminal number matches with the Current Symbol then the action is copied. If none matches then error is signalled.

NOTE:-

1. The entry '0' always matches with all terminal symbols.
2. If there is a shift on 'ERROR' symbol then it is always the first entry of the state. This has been done so that when we go down the stack looking for a state having a shift on 'ERROR', we have to check only the first entry of each state. This is needed during Panic mode error recovery.

Procedure PARSERUN drives the Parser. Given the current-state and current-symbol it calls GETACTION for the action to be taken. The configurations resulting after the different moves are as follows:-

a). SHIFT: The current-state & symbol are pushed into the stack and the Parser enters a new-state. The next symbol is got from the input.

b). REDUCE: 1). If we reduce by the production S' → S then we say that parsing is over.

2). If it is any other production then we pop the the stack 'l' elements, where 'l' is the length of the production. We then push the Non-Terminal which is on the R.H.S. of the production and the current-state onto the stack, and the parser enters a new state.

Semantic actions can be taken at this time.

c). ERROR: Call error recovery routine.
Procedure ERRORRECOVERY has to be written by the user.

3.3.10 Example:-

The computer output of the LALR Parser for the grammer G2 is shown on the next page.

```
      GRAMMER
      NONTERMINAL E T
      TERMINAL + * ID
      PRODUCTION
       E = E + T | E * T | T ; T = ID ; END

****STATE NO    1
[  1,   0]PREDS
          LALR  $


****STATE NO    2
[  5,   1]PREDS       1     5     6
          LALR  $           +           *


****STATE NO    3
[  1,   1]PREDS       1
          LALR  $

[  2,   1]
[  3,   1]

****STATE NO    4
[  4,   1]PREDS       1
          LALR  $           +           *


****STATE NO    5
[  2,   2]PREDS       1
          LALR  $           +           *


****STATE NO    6
[  3,   2]PREDS       1
          LALR  $           +           *


****STATE NO    7
[  2,   3]PREDS       1
          LALR  $           +           *


****STATE NO    8
[  3,   3]PREDS       1
          LALR  $           +           *

GOMAX=              4
MAXITEMS=           10SAVED=              8

   NTMAX=            3    TMAX=               6
   MAXP =            5    MAXSTATE=              8
```

```
*****TERMINALS
      1   NUL              0
      2   #                0
      3   ERROR            0
      4   +                0
      5   *                0
      6   ID               0

*****NONTERMINALS
       1   STARTP
       2   E
       3   T

*****PRODUCTIONS
      1   0   STARTP     =E          ;
      2   0   E          =E          +          T          ;
      3   0   E          =E          *          T          ;
      4   0   E          =T          ;
      5   0   T          =ID         ;
```

```
INITPROCEDURE;
BEGIN

PT[   1].A:=S ;PT[   1].T:=  6;PT[   1].NUM:=   2;
STATES[   1].PTR:=   1;STATES[   1].ENTRIES:=  1;

PT[   2].A:=R ;PT[   2].T:=  0;PT[   2].NUM:=   5;
STATES[   2].PTR:=   2;STATES[   2].ENTRIES:=  1;

PT[   3].A:=S ;PT[   3].T:=  5;PT[   3].NUM:=   6;
PT[   4].A:=S ;PT[   4].T:=  4;PT[   4].NUM:=   5;
PT[   5].A:=R ;PT[   5].T:=  2;PT[   5].NUM:=   1;
STATES[   3].PTR:=   3;STATES[   3].ENTRIES:=  3;

PT[   6].A:=R ;PT[   6].T:=  0;PT[   6].NUM:=   4;
STATES[   4].PTR:=   6;STATES[   4].ENTRIES:=  1;

PT[   7].A:=S ;PT[   7].T:=  6;PT[   7].NUM:=   2;
STATES[   5].PTR:=   7;STATES[   5].ENTRIES:=  1;

PT[   8].A:=S ;PT[   8].T:=  6;PT[   8].NUM:=   2;
STATES[   6].PTR:=   8;STATES[   6].ENTRIES:=  1;

PT[   9].A:=R ;PT[   9].T:=  0;PT[   9].NUM:=   2;
STATES[   7].PTR:=   9;STATES[   7].ENTRIES:=  1;

PT[  10].A:=R ;PT[  10].T:=  0;PT[  10].NUM:=   3;
STATES[   8].PTR:=  10;STATES[   8].ENTRIES:=  1;

END;

INITPROCEDURE;
BEGIN

NTGOTO[   1].PTR:=   1;NTGOTO[   1].NO:=   0;

TGOTO[   1].CS:=   1;TGOTO[   1].NS:=   3;
NTGOTO[   2].PTR:=   1;NTGOTO[   2].NO:=   1;

TGOTO[   2].CS:=   6;TGOTO[   2].NS:=   8;TGOTO[   3].CS:=   5;TGOTO[   3].NS:=   7;
TGOTO[   4].CS:=   1;TGOTO[   4].NS:=   4;
NTGOTO[   3].PTR:=   2;NTGOTO[   3].NO:=   3;


END;

INITPROCEDURE;
BEGIN
PRA[   1].NT:=   1;   PRA[   1].LEN:= 1; PRA[   2].NT:=   2;   PRA[   2].LEN:= 3;
PRA[   3].NT:=   2;   PRA[   3].LEN:= 3; PRA[   4].NT:=   2;   PRA[   4].LEN:= 1;
PRA[   5].NT:=   3;   PRA[   5].LEN:= 1;

END;

INITPROCEDURE;
BEGIN
TT[   1]:='NUL        ';TT[   2]:='(          ';TT[   3]:='ERROR      ';TT[   4]:='+
TT[   5]:='*          ';TT[   6]:='ID         ';
END;
```

CHAPTER-4

INSTRUCTIONS TO THE USER

This chapter gives information about how the Parser generator system can be used. Basically a Parser generator system consists of two Programs. The first Program called the CONSTRUCTOR generates the Parse Tables for the input grammer. The Second Program called the RECOGNIZER, is a set of routines which use the tables output by the 'Constructor' to parse sentences of the language. In our system the 'Constructor' program is the LALR.EXE file, and the 'Recognizer' program is the DRIVER.PAS file.

To use the system the user has to do the following:-

1. Give input grammer in the file named 'INPUT'. See appendix on how to specify the input grammer.
2. Run LALR.EXE
3. In case errors are reported by the program then examine the output file 'OUTPUT' for the errors. See the section on error messages for the errors reported by the system.
4. If no errors are reported then the Program writes MAXITEMS = 'number' and SAVED = 'number' on the TTY. The value of SAVED is just an indication to the user about how much the table has been compacted. This value is not used elsewhere.

In the file named 'OUTPUT' the values of the constants GOMAX, MAXITEMS, NTMAX, TMAX, MAXP and MAXSTATE are given. These constants are the values of various table sizes that have been output by the program. These values have to be entered in the CONST declaration section of the DRIVER.PAS file.

5. Add the Lexical analysis, error Recovery and semantic routines in the DRIVER.PAS file.

6. Copy the file 'PARSET' outputted by the previous program, after the declaration section of the DRIVER.PAS program.

7. Compile and execute the DRIVER.PAS file.

The following sections provide details about the system.

4.1 Output Files:-

The LALR.EXE program outputs two files named 'OUTPUT' and 'PARSET'.

4.1.1. The file 'OUTPUT' contains the following information:-

a) The Input grammer given by the user. In case errors are detected, the error positions are marked and suitable error numbers are written. At the end of the file, for each error number a suitable error message is given.

b) If there are no errors in the Input grammer, then the states generated will be listed. For all productions in the state, which are complete or are of the form ($A \rightarrow \varphi . B\delta$ and $\delta \overset{*}{\Rightarrow} e$), the predecessor states PREDS and the LALR set computed will be enumerated.

A typical state listing is given below:-

```
***** State  no. 10
[2,1 ]
[3,1 ]
[3,3 ]  PREDS     1        3        6        7
        LALR      +        *
```

[2,1] [3,1] tells us that the productions 2 and 3 are in this state and the dot is after the first position. For [3,1] which is a completed item the PREDS and LALR have been listed.

LALR (+ , *) tells us that these are the lookahead symbols that have been computed for this production. PREDS (1,3,6,7) tells us that these lookaheads have come from items in states 1,3,6 and 7.

c) Constant values which give the size of various tables generated by the program. These values have to be entered in the 'const declaration part' of the DRIVER.PAS file.

d) In case there are Shift-Reduce or Reduce-Reduce conflicts in the Parse Table, then the states in which the conflict has occured are listed. See section 4.4 for error messages.

e) The terminals, Non-Terminals and Productions are outputted alongwith numbers assigned to them in the program.

The Terminals are listed as follows:-

```
4     +     29  LEFT
6     ID    0
```

This tells us that the Terminal symbol '+' has been assigned number 4, its precedence is 29 and associativity is LEFT. If associativity is not defined then nothing is written in this field. The default value of Precedence is 0. Terminals 'NUL', '$', 'ERROR' are standard names and are always outputted. Users can't use these names for their Terminals.

The Productions are listed as follows:-

```
5    29    E = E    +    T;
```

This tells us that the Production number is 5, its Precedance is 29. In case Precedence for a Production is not defined then it is assumed to be Zero.

STARTP =S is a standard Production used to augment the grammer and is always the first production listed.

4.1.2 The file 'PARSET' contains the Parse table. It is a series of INITPROCEDURE declarations. This is because PASREL doesn't handle very large procedures. This file should be copied after the declaration section of the DRIVER.PAS file.

4.2 Lexical Analyzer:-

Lexical Analyzer recognizes tokens in the Input and passes them to the Parser. This routine has to be written by the user. The procedure name has to be INSYMBOL, as the Parser calls lexical analyzer by this name. The value has to be passed in the variable 'CSYMBOL'. A number is associated to each Terminal defined by the user. When a terminal symbol is recognized in the input, the number associated with the symbol should be passed in 'CSYMBOL'. The numbers associated with the Terminal symbols are listed in the output file. For end of file the value 2 has to be passed as this number is assigned for end of the file.

4.3 Semantic Actions:-

Semantic Actions can be incorporated in LR Parsing action tables. However, it is customary in "bottom-up" compilers to associate semantic actions only with the reduce actions, thereby allowing semantic modules to be cleanly separated from the Parsing module. Since LR parsers have no difficulty with empty right sides, null productions can be inserted anywhere in the grammer as hooks on which to hang semantics.

The reader is referred to Chapter 7 of [1] for more details on Semantic analysis.

Thus the Semantic analysis phase can be totally separated from the Parsing phase. The output of Parsing phase is a list of Production numbers in the order they were reduced. In case transliteration of the input has been done then the actual symbols have also to be outputted alongwith the Production number.

4.4 Error Messages:-

Following are the error messages given on the TTY by the LALR.EXE Program:

a) 'Program Aborted': This error is given when there are errors in the Input Grammer. The error listing is given in the file called 'OUTPUT'.

b) 'NT UNDECIDED': This error is given when it is not possible for procedure 'LAMBDA' to determine whether the non-terminal number given generates the Null string or not. This generally indicates that the input grammer is wrong and some productions have not been written.

c) '(Constant Name) Exceeded': This error message is given when the array bounds are exceeded. For e.g. 'NTMAX EXCEEDED' means that maximum number of Non-Terminals allowed have been exceeded. All these constant values are declared in the CONST declarations part of the program LALR.PAS. The user should increase the constant value exceeded and then recompile LALR.PAS file. The

commands are as follows:-

.R PASREL
* LALR.PAS
. LOAD LALR.REL
. SAVE LALR.EXE /40

NOTE:- Maximum value of NTMAX is 210 and TMAX is 140. These values can not be changed.

d) 'SHIFT REDUCE ERRORS': This error message is given when there are conflicts in the Parsing table which could not be resolved. This means that the grammer is not LALR(1). The errors are listed in the 'OUTPUT' file. Errors are reported in the following form:

***** Errors in state = 5

Shift Reduce error. TNO = 6 PROD NO = 14
Reduce Reduce error. TNO = 7 PROD NO = 15, PROD NO = 17

The above message tells us that in state 5, a shift Reduce conflict has occured on Terminal number 6. The Productic number involved is 14. Similarly for the second case both the productions 15 and 17 are trying to reduce on Terminal number 7. The user has to modify the input grammer or resolve the conflicts by giving PRECEDENCE information. To get more idea about the error and how it has been generated, the user can see all the items in the state, the LALR lookaheads and Predecessor states.

4.5 Handling Ambiguous Grammers:

It is a theorem that every ambiguous grammer fails to be LR. However the user can specify ambiguous grammers and resolve the conflict by giving Precedence and associativity declarations. This is needed for

instance to solve the 'dangling else' problem in Pascal. For a good description on handling of ambiguous grammer, the user is referred to chapter 6 of [1].

Precedence can be assigned to all Terminals. The Terminals listed first are given highest precedence (number 30) and others are given lower precedence. For e.g. the declarations

```
Precedence   left   *
Precedence   left   +   -
Precedence   else
Precedence   then
```

Thus * has precedence 30 and is left associative.
+,- have precedence 29 and are left associative.
'else' has precedence 28 and associativity is undefined
'then' has precedence 27 and associativity is undefined.

The Productions can be assigned precedence of any of the Terminals given in the precedence declaration. For e.g.

```
3  Stmt =  if  C  then Stmt Precedence then
4  Stmt =  if  C  then Stmt else Stmt Precedence else
```

If we have shift-reduce conflict on the Terminal 'else' with Prod. no. 3 then the Parser resolves the conflict in favour of shift action because the Precedence of terminal 'else' is 28 and Precedence of Prod. no. 3 is 27. (= the Precedence of Terminal 'then').

4.6 Error Recovery:-

Most of the existing automatic error recovery methods are quite

poor in performance. Besides too much restriction is imposed on the way productions have to be written by the user. The error recovery method given here is the one used in YACC and also discussed in [1,4] . The error routines have to be written by the user.

The user identifies major non-terminals of his program and writes the error productions of the type "$A \rightarrow \alpha$ error." (error is a standard terminal name). In the previous chapter we have described that the parser compacts the table by using 'default reductions' in states. These however severely hamper local error recovery as reduction can occur on wrong inputs. The Parser enumerates all lookaheads in states which have a shift on 'error'. Then the user can try local error recovery by deleting, inserting or replacing the input symbols by one of the shift symbols of that state.

In case local recovery is not possible then we go down the parse stack till a shift on 'error' is encountered. The parser then shifts over the 'error' token and then reduces by the error production. Then the appropriate error recovery routine is called which skips the input till a shift symbol of the current stack is encountered. Besides certain major keywords of the language are never skipped. In case these are encountered the parse stack has to be reconfigured. Method to reconfigure the stack for LL(1) parser has been discussed in [7] and with some modifications it can be adopted for the LALR(1) parser as well.

4.7 Examples:

The computer outputs for grammer G1 are given on the next page. Note the errors reported in the OUTPUT file because the grammer is ambiguous.

Grammer G1 is then modified by giving Precedence and associativity declarations to Terminals and Productions. The output of the modified G1 Grammer is also given.

```
      GRAMMER
      NONTERMINAL E
      TERMINAL + * ID
      PRODUCTION
        E = E + E | E * E | ID ;
      END

****STATE NO   1
[  1,  0]PREDS
         LALR  $

****STATE NO   2
[  4,  1]PREDS      1     4     5
         LALR  $          +           *

****STATE NO   3
[  1,  1]PREDS      1
         LALR  $

[  2,  1]
[  3,  1]

****STATE NO   4
[  2,  2]PREDS      1     4     5
         LALR  $          +           *

****STATE NO   5
[  3,  2]PREDS      1     4     5
         LALR  $          +           *

****STATE NO   6
[  2,  1]
[  2,  3]PREDS      1     4     5
         LALR  $          +           *

[  3,  1]

****STATE NO   7
[  2,  1]
[  3,  1]
[  3,  3]PREDS      1     4     5
         LALR  $          +           *

*****ERRORS IN STATE=   6
  SHIFT REDUCE ERROR,TNO=  4PROD NO=   2
  SHIFT REDUCE ERROR,TNO=  5PROD NO=   2
```

CENTRAL LIBRARY
Kanpur.
No. A 82759

```
*****ERRORS IN STATE=    7
  SHIFT REDUCE ERROR.TNO=   4PROD NO=    3
  SHIFT REDUCE ERROR.TNO=   5PROD NO=    3
GOMAX=          3
MAXITEMS=          13SAVED=          2

   NTMAX=           2    TMAX=            6
   MAXP =           4    MAXSTATE=            7

*****TERMINALS
      1   NUL          0
      2   $            0
      3   ERROR        0
      4   +            0
      5   *            0
      6   ID           0

*****NONTERMINALS
       1   STARTP
       2   E

*****PRODUCTIONS
     1   0   STARTP     =E          ;
     2   0   E          =E          +          E          ;
     3   0   E          =E          *          E          ;
     4   0   E          =ID         ;
```

```
INITPROCEDURE;
BEGIN

PT[   1].A:=S ;PT[   1].T:=  6;PT[   1].NUM:=  2;
STATES[   1].PTR:=   1;STATES[   1].ENTRIES:=  1;

PT[   2].A:=R ;PT[   2].T:=  0;PT[   2].NUM:=  4;
STATES[   2].PTR:=   2;STATES[   2].ENTRIES:=  1;

PT[   3].A:=S ;PT[   3].T:=  5;PT[   3].NUM:=  5;
PT[   4].A:=S ;PT[   4].T:=  4;PT[   4].NUM:=  4;
PT[   5].A:=R ;PT[   5].T:=  2;PT[   5].NUM:=  1;
STATES[   3].PTR:=   3;STATES[   3].ENTRIES:=  3;

PT[   6].A:=S ;PT[   6].T:=  6;PT[   6].NUM:=  2;
STATES[   4].PTR:=   6;STATES[   4].ENTRIES:=  1;

PT[   7].A:=S ;PT[   7].T:=  6;PT[   7].NUM:=  2;
STATES[   5].PTR:=   7;STATES[   5].ENTRIES:=  1;

PT[   8].A:=S ;PT[   8].T:=  5;PT[   8].NUM:=  5;
PT[   9].A:=S ;PT[   9].T:=  4;PT[   9].NUM:=  4;
PT[  10].A:=R ;PT[  10].T:=  2;PT[  10].NUM:=  2;
STATES[   6].PTR:=   8;STATES[   6].ENTRIES:=  3;

PT[  11].A:=S ;PT[  11].T:=  5;PT[  11].NUM:=  5;
PT[  12].A:=S ;PT[  12].T:=  4;PT[  12].NUM:=  4;
PT[  13].A:=R ;PT[  13].T:=  2;PT[  13].NUM:=  3;
STATES[   7].PTR:=   11;STATES[   7].ENTRIES:=  3;

END;

INITPROCEDURE;
BEGIN

NTGOTO[   1].PTR:=   1;NTGOTO[   1].NO:=   0;

TGOTO[   1].CS:=   5;TGOTO[   1].NS:=   7;TGOTO[   2].CS:=   4;TGOTO[   2].NS:=   6;
TGOTO[   3].CS:=   1;TGOTO[   3].NS:=   3;
NTGOTO[   2].PTR:=   1;NTGOTO[   2].NO:=   3;


END;

INITPROCEDURE;
BEGIN
PRA[   1].NT:=  1;  PRA[   1].LEN:= 1; PRA[   2].NT:=  2;  PRA[   2].LEN:= 3;
PRA[   3].NT:=  2;  PRA[   3].LEN:= 3; PRA[   4].NT:=  2;  PRA[   4].LEN:= 1;

END;

INITPROCEDURE;
BEGIN
TT[   1]:='NUL         ';TT[   2]:='(           ';TT[   3]:='ERROR       ';TT[   4]:='+
TT[   5]:='*           ';TT[   6]:='ID          ';
```

```
      GRAMMER
      NONTERMINAL E
      TERMINAL + * ID
      PRECEDENCE LEFT +
      PRECEDENCE LEFT *
      PRODUCTION
       E = E + E PRECEDENCE + ! E * E PRECEDENCE * ! ID ;
      END

****STATE NO    1
[  1,   0]PREDS
          LALR  6


****STATE NO    2
[  4,   1]PREDS      1      4      5
          LALR  6            +            *


****STATE NO    3
[  1,   1]PREDS      1
          LALR  6

[  2,   1]
[  3,   1]

****STATE NO    4
[  2,   2]PREDS      1      4      5
          LALR  6            +            *


****STATE NO    5
[  3,   2]PREDS      1      4      5
          LALR  6            +            *


****STATE NO    6
[  2,   1]
[  2,   3]PREDS      1      4      5
          LALR  6            +            *

[  3,   1]

****STATE NO    7
[  2,   1]
[  3,   1]
[  3,   3]PREDS      1      4      5
          LALR  6            +            *

GOMAX=              3
MAXITEMS=             11SAVED=              4

   NTMAX=               2   TMAX=                6
   MAXP =               4   MAXSTATE=                7
```

```
*****TERMINALS
      1   NUL            0
      2   $              0
      3   ERROR          0
      4   +             30  LEFT
      5   *             29  LEFT
      6   ID             0

*****NONTERMINALS
      1   STARTP
      2   E

*****PRODUCTIONS
      1   0   STARTP     =E          ;
      2  30   E          =E          +          E          ;
      3  29   E          =E          *          E          ;
      4   0   E          =ID         ;
```

```
INITPROCEDURE;
BEGIN

PT[   1].A:=S ;PT[   1].T:=  6;PT[   1].NUM:=   2;
STATES[  1].PTR:=   1;STATES[  1].ENTRIES:=  1;

PT[   2].A:=R ;PT[   2].T:=  0;PT[   2].NUM:=   4;
STATES[  2].PTR:=   2;STATES[  2].ENTRIES:=  1;

PT[   3].A:=S ;PT[   3].T:=  5;PT[   3].NUM:=   5;
PT[   4].A:=S ;PT[   4].T:=  4;PT[   4].NUM:=   4;
PT[   5].A:=R ;PT[   5].T:=  2;PT[   5].NUM:=   1;
STATES[  3].PTR:=   3;STATES[  3].ENTRIES:=  3;

PT[   6].A:=S ;PT[   6].T:=  6;PT[   6].NUM:=   2;
STATES[  4].PTR:=   6;STATES[  4].ENTRIES:=  1;

PT[   7].A:=S ;PT[   7].T:=  6;PT[   7].NUM:=   2;
STATES[  5].PTR:=   7;STATES[  5].ENTRIES:=  1;

PT[   8].A:=R ;PT[   8].T:=  0;PT[   8].NUM:=   2;
STATES[  6].PTR:=   8;STATES[  6].ENTRIES:=  1;

PT[   9].A:=R ;PT[   9].T:=  5;PT[   9].NUM:=   3;
PT[  10].A:=S ;PT[  10].T:=  4;PT[  10].NUM:=   4;
PT[  11].A:=R ;PT[  11].T:=  2;PT[  11].NUM:=   3;
STATES[  7].PTR:=   9;STATES[  7].ENTRIES:=  3;

END;

INITPROCEDURE;
BEGIN

NTGOTO[  1].PTR:=   1;NTGOTO[  1].NO:=   0;

TGOTO[   1].CS:=   5;TGOTO[   1].NS:=   7;TGOTO[   2].CS:=   4;TGOTO[   2].NS:=   6;
TGOTO[   3].CS:=   1;TGOTO[   3].NS:=   3;
NTGOTO[  2].PTR:=   1;NTGOTO[  2].NO:=   3;


END;

INITPROCEDURE;
BEGIN
PRA[  1].NT:=  1;  PRA[  1].LEN:= 1; PRA[  2].NT:=  2;  PRA[  2].LEN:= 3;
PRA[  3].NT:=  2;  PRA[  3].LEN:= 3; PRA[  4].NT:=  2;  PRA[  4].LEN:= 1;

END;

INITPROCEDURE;
BEGIN
TT[  1]:='NUL         ';TT[  2]:='              ';TT[  3]:='ERROR     ';TT[  4]:='+
TT[  5]:='*           ';TT[  6]:='ID            ';
END;
```

CHAPTER-5

CONCLUSIONS

The LALR(1) Parser generator system described in this report has been tested on various input grammers. We have tested it on the Pascal grammer, which had about 150 productions, 60 terminals and 60 Non-terminals.

The Program currently outputs the actions for a state in a list form. The list consists of pairs of a terminal symbol and the action associated with the symbol. This means that we have to search the list for the Terminal Symbol to get the action. Though this method uses less storage space, it may be slow. Thus a different scheme for table representation can be implemented which makes the searching faster. One such technique, could be that the list is sorted, so that the most frequently occuring symbols are near the head of the list. Other techniques like matrix representation could also be used. These changes can be made without affecting the rest of the Program.

Not much has been done for error recovery in this implementation. Though it is difficult to automatic error recovery fully, some parts could be possibly automated. For instance the Panic mode error recovery could be possibly automated. Error recovery schemes have to be incorporated in the Driver program.

Automatic Parser generators are only a small part of the Translator Writing Systems and much work needs to be done, in case we want to automate the next phases of compiler writing. The output of the Parser should be suitably defined, so that it is compatible with the input needed by the Semantic phase. These changes have to be made in the Driver routine.

APPENDIX

INPUT GRAMMER

KEYWORDS: GRAMMER TERMINAL NONTERMINAL PRECEDENCE PRODUCTION
END LEFT RIGHT NONASSOC NUL

OPERATORS: = ; |

Lexical Conventions:

1) Blanks are used as delimiters for all lexical tokens. Hence blanks should be always put between tokens.

2) IDENTIFIER is

a) Sequence of letters

or b) Sequence of characters which are not in letters or operators.

or c) a string of characters in quotes.

Thus following are valid identifiers:

NAME 'END' ';' '>='

NOTE: END ; >= are put in quotes because END is a keyword and ; >= contains operators ; and = .

Grammer Specification:

The Input grammer is described below:

```
usergrammer = "GRAMMER" ntdecl tdecl prdecl proddecl "END"
ntdecl      = "NONTERMINAL" ntlist
tdecl       = "TERMINAL" idlist
prdecl      = NUL | "PRECEDENCE" prtype idlist prdecl
```

```
prtype      =   NUL | "LEFT" | "RIGHT" | "NONASSOC"

proddecl    =   "PRODUCTION"    plist

plist       =   ntid    "="     rhspart
            |   plist   ntid    "="     rhspart

rhspart     =   idlist  precedencepart  ";"
            |   idlist  precedencepart  "|"     rhspart

precedencepart = NUL | "PRECEDENCE"     IDENTIFIER

ntlist      =   ntid | ntlist   ntid

ntid        =   IDENTIFIER

idlist      =   IDENTIFIER | idlist     IDENTIFIER
```

Note: a) "ntid" checks that the identifier has only letters. Non-Terminal names should use only letters.

b) NUL, $, ERROR are pre-declared terminal names.

c) STARTP is a pre-declared Non-Terminal name.

d) The first identifier declared in the Non-Terminal list is taken as the START symbol of the grammer.

REFERENCES:

1. Aho A.V. & Ullman J.D.: Principles of Compiler Design

2. Bauer F.L. & Eickel J: Compiler Construction (An Advanced Course). pp 85-107

3. DeRemer Frank, Penello T.T.: Efficient Computation of LALR(1) lookahead sets. Sigplan Notices Vol 14, No. 8, Aug 1979 pp 176-187.

4. Graham S.L. et.al.: Practical LR Error Recovery. Sigplan Notices Vol 14, No. 8, Aug 1979 pp 168-175.

5. Kristensen B.B. & Madsen O.L.: Method for Computing LALR(K) lookahead. TOPLAS, Jan'81, Vol 3, No 1, pp 60-82.

6. Kristensen B.B. & Madsen O.L.: Diagnostics on LALR(K) Conflicts based on a method for LR(K) testing. BIT 21 (1981), pp 270-293.

7. Pai A.B., Kieburtz R.B.: Global Context Recovery: A new strategy for recovery from Syntax errors. Sigplan Notices, Vol 14, No. 8, Aug 1979 pp 158-167

```
PROGRAM LALR(INPUT,OUTPUT,PARSEFILE);
  label
      1;
      const PRODMAX=200;(* MAX NO. OF PRDUCTIONS *)
      MAXPROD=800;(* MAX LENGTH OF PRODUCTIONS*)
      TMAX   =140;(* MAX NO OF TERMINALS *)
      DONEMAX=300;
      NTMAX  =140;(* MAX NO OF NONTERMINALS *)
      TANDNTMAX=140;(* MAX. OF TMAX AND NTMAX *)
      MAXSTATES=500;(* MAX NO OF STATES *)
      MAXITEMS=1000;
      MAXTSIZE=1500;
      MAXKERNELitems=50;
      PRMAX=30;(* NO OF PRECEDENCES*)
      ALNG=10;(* MAX CHAR IN ID *)
      LINELENG=131;
      NKW=9;(* NO OF KEYWORDS*)
  type
      FILENAME=packed array[1..9] of char;
      ALPHASTYPE=(TERMINAL,NONTERMINAL);
      ACTIONTYPE=(SHIFT,REDUCE);
      ACTTYPE=(SH,RE);
      ALPHATYPE=packed record
                                ALPHA:ALPHASTYPE;
                                NUM:integer
                              end;
      GOTOTYPE=^GOTOTYPE1;
      FIRSTSET=record
                        H,L:set of 1..70
                end;
      LASET=FIRSTSET;
      NULPTR=^NULP;
      NULP=record
                  PRODNO:integer;
                  LALR:LASET;NEXTNUL:NULPTR
           end;
      GOTOTYPE1=packed record
                                SYMBOL:1..TMAX;
                                NEXTSTATE:integer;(* IN CASE OF REDUCE IT IS PRODNO*)
                                ACTION:ACTIONTYPE;
                                PTRN:GOTOTYPE
                              end;
      BGOTO=^AGOTO;
      AGOTO=packed record
                              CURRENTSTAte,NEXTSTATE:0..MAXSTATES;
                              PTR:BGOTO
                    end;
      PREPTR=^PREDSTATE;
      PREDSTATE=record
                          STATENO:integer;
                          NPTR:PREPTR
```

```
                end;
    KERNELITEM=packed record
                              LAM:boolean;(* TRUE IF PROD GIVES LAMDA *)
                              LADONE:boolean;(* TRUE IF LALR HAS BEEN COMPUTED *)
                              PRODNO:integer;
                              PLACE:integer;
                              PREDPTR:PREPTR;
                              LALR:LASET
                            end;
    SYMBOL=(TSYM,NTSYM,GRSYM,PRSYM,PSYM,USYM,RSYM,LSYM,NONSYM,
            ENDSYM,BARSYM,SEMICOLON,EOSYM,IDSYM,IDTSYM);
    SYMSET=set of SYMBOL;
    ALFA=packed array [1..ALNG] of char;
    MSGS=packed array [1..20] of char;
var
    PARSEFILE:file of char;
    PARSENAME:FILENAME;
    FMSG: array [1..9] of MSGS;
    KEY:array [0..NKW] of ALFA;
    KSY:array [1..NKW] of SYMBOL;
    MSG:array[0..9] of MSGS;
    PROD:array[1..MAXPROD]of ALPHATYPE;
    PRODARRAY:array [1..PRODMAX] of record
                                      PRECEDENCE:integer;
                                      NT:integer;
                                      NEXTPROD:integer;
                                      PRODPTR:integer;
                                      LENGTH:integer
                                    end;
    NTARRAY:array [0..NTMAX]  of integer;
    NTSONS:array[0..NTMAX] of integer;
    STATES:array [1..MAXSTATES] of record
                                     NOOFITEMS:integer;
                                     KPTR:integer;(*POINTS IN ARRAY WHERE KERNEL BEGI
                                     NPTR:integer;(*USED BY SEARCH*)
                                     PGOTO:GOTOTYPE;
                                     NULPROD:NULPTR
                                   end;
    ITEMS:array [1..MAXITEMS] of KERNELITEM;
    TGOTO:array [1..NTMAX] of PGOTO;
    STATECOUNT,CSTATE:integer;
    ITEMPTR:integer;
    LAMDA:array [1..NTMAX] of boolean;
    FT:array[1..NTMAX] of FIRSTSET;
    NTSAVE:array [0..NTMAX] of ALFA;
    TSAVE:array [0..TMAX] of ALFA;
    TPRED:array[1..TMAX] of record
                              PRECEDENCE:integer;
                              ASSOCSYM:SYMBOL
                            end;
    NTCOUNT,PCOUNT,TCOUNT,MAXP,PRCOUNT:integer;
```

```
    TEMP1:array [0..TMAX] of integer;
    TEMP2:array [0..MAXSTATES] of record
                                      PTR,NEXT:integer
                                  end;
    TPARSE:array [1..MAXTSIZE] of packed record
                                      ACTION:ACITYPE;
                                      T:0..TMAX;
                                      NUM:0..MAXSTATES
                                  end;
    INITCOUNT,TEMP2PTR:integer;
initprocedure;
      begin KEY[1]:='GRAMMER   ';
            KEY[2]:='TERMINAL  ';
            KEY[3]:='NONTERMINA';
            KEY[4]:='PRECEDENCE';
            KEY[5]:='PRODUCTION';
            KEY[6]:='RIGHT     ';
            KEY[7]:='LEFT      ';
            KEY[8]:='NONASSOC  ';
            KEY[9]:='END       ';
      end;
initprocedure;
      begin
            EMSG[1]:='TOO MANY TERMINALS   ';
            EMSG[2]:='TOO MANY NONTERMINAL ';
            EMSG[3]:='TOO MANY PRECEDENCES ';
            EMSG[4]:='TOO MANY PRODUCTIONS ';
            EMSG[5]:='MAXPROD EXCEEDED     ';
            EMSG[6]:='MAXKERNELITEM EXCEED ';
            EMSG[7]:='MAXSTATES EXCEEDED   ';
            EMSG[8]:='MAXITEMS EXCEEDED    ';
            EMSG[9]:='DONEMAX EXCEEDED     ';
      end;
initprocedure;
      begin
            KSY[1]:=GRSYM;KSY[2]:=TSYM;KSY[3]:=NTSYM;
            KSY[4]:=PSYM;KSY[5]:=PRSYM;KSY[6]:=RSYM;
            KSY[7]:=LSYM;KSY[8]:=NONSYM;KSY[9]:=ENDSYM;
            MSG[0]:='ID  EXPECTED          ';
            MSG[1]:='GRAMMER     EXPECTED';
            MSG[2]:='TERMINAL    EXPECTED';
            MSG[3]:='NONTERMINAL EXPECTED';
            MSG[4]:='PRECEDENCE  EXPECTED';
            MSG[5]:='PRODUCTION EXPECTED ';
            MSG[6]:='ASSOCIATION EXPECTED';
            MSG[7]:='= EXPECTED            ';
            MSG[8]:='END    EXPECTED       ';
            MSG[9]:=' ; !    EXPECTED      ';
      end;
procedure FATALERROR(I:integer);
      begin WRITELN(TTY,EMSG[I]);goto 1
```

```
    end;
procedure READINPUT;
    const
        ERRMAX=10;
    var
        SY:SYMBOL;      ID:ALFA;
        CH:char;CC,LL:integer;
        ERRS:set of 0..ERRMAX;
        ERRPOS:integer;
        LINE:array [1..LINELENG] of char;
        NUL:boolean;
        I:integer;
        FSYS:SYMSET;
    procedure INSYMBOL;
        forward;
    procedure ERRORMSG;
        forward;
    procedure NEXTCH;
        var
            MYTEMP:integer;
        begin
            if CC=LL then
              begin
                if EOF(INPUT) then
                  begin WRITELN;WRITELN('INCOMPLETE GRAMMER');ERRORMSG;goto 1
                  end;
                if ERRPOS <>0 then
                  begin WRITELN;ERRPOS:=0
                  end;
                LL:=0;CC:=0;WRITE('     ');
                while not EOLN(INPUT) do
                  begin LL:=LL+1;READ(CH);WRITE(CH);
                      MYTEMP:=ORD(CH);
                      if (MYTEMP >= 97) then
                          if (MYTEMP <=122)  then CH:=CHR(MYTEMP-32);
                      LINE[LL]:=CH
                  end;
                WRITELN; LL:=LL+1;READ(LINE[LL])
              end;
            CC:=CC+1;CH:=LINE[CC]
        end;
    procedure ERRORMSG;
        var
            K:integer;
        begin K:=0;while ERRS <> [] do
              begin while not (K in ERRS)do K:=K+1;
                  WRITELN;WRITELN(K,' ',MSG[K]);ERRS:=ERRS-[K]
              end
        end;
    procedure ERROR(N:integer);
        begin
```

```
    end;
procedure READINPUT;
    const
        ERRMAX=10;
    var
        SY:SYMBOL;      ID:ALFA;
        CH:char;CC,LL:integer;
        ERRS:set of 0..ERRMAX;
        ERRPOS:integer;
        LINE:array [1..LINELENG] of char;
        NUL:boolean;
        I:integer;
        FSYS:SYMSET;
    procedure INSYMBOL;
        forward;
    procedure ERRORMSG;
        forward;
    procedure NEXTCH;
        var
            MYTEMP:integer;
        begin
            if CC=LL then
              begin
                if EOF(INPUT) then
                  begin WRITELN;WRITELN('INCOMPLETE GRAMMER');ERRORMSG;goto 1
                  end;
                if ERRPOS <>0 then
                  begin WRITELN;ERRPOS:=0
                  end;
                LL:=0;CC:=0;WRITE('     ');
                while not EOLN(INPUT) do
                  begin LL:=LL+1;READ(CH);WRITE(CH);
                        MYTEMP:=ORD(CH);
                        if (MYTEMP >= 97) then
                              if (MYTEMP <=122)  then CH:=CHR(MYTEMP-32);
                        LINE[LL]:=CH
                  end;
                WRITELN; LL:=LL+1;READ(LINE[LL])
              end;
            CC:=CC+1;CH:=LINE[CC]
        end;
    procedure ERRORMSG;
        var
            K:integer;
        begin K:=0;while ERRS <> [] do
              begin while not (K in ERRS)do K:=K+1;
                    WRITELN;WRITELN(K,' ',MSG[K]);ERRS:=ERRS-[K]
              end
        end;
    procedure ERROR(N:integer);
        begin
```

```
            if ERRPOS = 0 then WRITE('     ');
            if CC > ERRPOS then
              begin WRITE (' ':CC-ERRPOS,'^',N:1);
                ERRPOS:=CC+1;ERRS:=ERRS+[N]
              end
        end;
procedure SKIP (FSYS:SYMSET;N:integer);
        begin ERROR(N);
            while not (SY in FSYS) do INSYMBOL
        end;
procedure TEST(S1,S2:SYMSET;N:integer);
        begin
            if not (SY in S1) then SKIP (S1+S2,N)
        end;
procedure INSYMBOL;
        label
            2,3;
            var K:integer;
        begin K:=0;ID:='          ';
            while CH=' ' do NEXTCH;
            if CH in ['A'..'Z'] then
              begin
                  repeat
                        if K < ALNG then
                          begin K:=K+1;ID[K]:=CH
                          end;
                        NEXTCH
                  until not (CH in ['A'..'Z']);
                  KEY[0]:=ID;K:=NKW;
                  while KEY[K] <> ID do K:=K-1;
                  if K=0 then SY:=IDSYM
                  else SY:=KSY[K]
              end
            else
            case CH of
                  ';':
                        begin SY:=SEMICOLON;NEXTCH
                        end;
                  '=':
                        begin SY:=EQSYM;    NEXTCH
                        end;
                  '!':
                        begin SY:=BARSYM;NEXTCH
                        end;
                  '''':
                        begin
                            2:
                            NEXTCH;
                            if CH='''' then
                              begin NEXTCH;
                                  if CH <> '''' then goto 3
```

```
                                end;
                            if K < ALNG then
                              begin K:=K+1;ID[K]:=CH
                              end;
                            goto 2;
                            3:
                            SY:=IDTSYM
                        end;
                others:
                        begin
                            repeat
                                if K < ALNG then
                                  begin K:=K+1;ID[K]:=CH
                                  end;
                                NEXTCH
                            until (CH in ['A'..'Z'])or(CH=' ');
                            SY:=IDTSYM
                        end
            end;
        end;
procedure TERMINALS(S,S1:SYMSET);
        begin INSYMBOL;
            TEST([IDSYM,IDTSYM],S+S1,0);
            while (SY=IDSYM)or(SY=IDTSYM) do
              begin TCOUNT:=TCOUNT+1;
                    if TCOUNT > TMAX then FATALERROR(1);
                    TSAVE[TCOUNT]:=ID;INSYMBOL;
                    with TPRED[TCOUNT] do
                      begin PRECEDENCE:=0;ASSOCSYM:=USYM
                      end;
                    TEST([IDSYM,IDTSYM]+S,S1,0)
              end;
        end;
procedure NONTERMS(S,S1:SYMSET);
        begin INSYMBOL;TEST([IDSYM],S+S1,0);
            while SY=IDSYM do
              begin NTCOUNT:=NTCOUNT+1;
                    if NTCOUNT > NTMAX then FATALERROR(2);
                    NTSAVE[NTCOUNT]:=ID;INSYMBOL;
                    TEST([IDSYM]+S,S1,0)
              end;
        end;
procedure SEARCHT(var I:integer);
        begin TSAVE[0]:=ID; I:=TCOUNT;
            while TSAVE[I] <> ID do I:=I-1
        end;
procedure SEARCHNT(var I:integer);
        begin NTSAVE[0]:=ID;I:=NTCOUNT;
            while NTSAVE[I] <> ID do I:=I-1;
        end;
procedure SEARCHTNT(var K:integer;var X:ALPHATYPE);
```

```
                                end;
                              if K < ALNG then
                                begin K:=K+1;ID[K]:=CH
                                end;
                              goto 2;
                              3:
                              SY:=IDTSYM
                            end;
                    others:
                            begin
                              repeat
                                if K < ALNG then
                                  begin K:=K+1;ID[K]:=CH
                                  end;
                                NEXTCH
                              until (CH in ['A'..'Z'])or(CH=' ');
                              SY:=IDTSYM
                            end
            end;
        end;
procedure TERMINALS(S,S1:SYMSET);
        begin INSYMBOL;
            TEST([IDSYM,IDTSYM],S+S1,0);
            while (SY=IDSYM)or(SY=IDTSYM) do
              begin TCOUNT:=TCOUNT+1;
                if TCOUNT > TMAX then FATALERROR(1);
                TSAVE[TCOUNT]:=ID;INSYMBOL;
                with TPRED[TCOUNT] do
                  begin PRECEDENCE:=0;ASSOCSYM:=USYM
                  end;
                TEST([IDSYM,IDTSYM]+S,S1,0)
              end;
        end;
procedure NONTERMS(S,S1:SYMSET);
        begin INSYMBOL;TEST([IDSYM],S+S1,0);
            while SY=IDSYM do
              begin NTCOUNT:=NTCOUNT+1;
                if NTCOUNT > NTMAX then FATALERROR(2);
                NTSAVE[NTCOUNT]:=ID;INSYMBOL;
                TEST([IDSYM]+S,S1,0)
              end;
        end;
procedure SEARCHT(var I:integer);
        begin TSAVE[0]:=ID; I:=TCOUNT;
            while TSAVE[I] <> ID do I:=I-1
        end;
procedure SEARCHNT(var I:integer);
        begin NTSAVE[0]:=ID;I:=NTCOUNT;
            while NTSAVE[I] <> ID do I:=I-1;
        end;
procedure SEARCHTNT(var K:integer;var X:ALPHATYPE);
```

```
        begin NUL:=false;SEARCHNT(K);
            if K # 0 then
              begin X.ALPHA:=NONTERMINAL;X.NUM:=K
              end
            else
              begin SEARCHT(K);
                  if K # 0 then
                        if K=1 then NUL:=true
                        else
                          begin X.ALPHA:=TERMINAL;X.NUM:=K
                          end;
              end;
        end;
procedure PRECEDENCE(S,S1:SYMSET);
        var
            I,K:integer;
            TSYM:SYMBOL;
        begin I:=PRMAX+1;
            while SY = PSYM do
              begin INSYMBOL;
                  if SY in [LSYM,RSYM,NONSYM] then
                    begin TSYM:=SY;INSYMBOL
                    end
                  else TSYM:=USYM;
                  while (SY=IDSYM) or (SY=IDTSYM) do
                    begin I:=I-1;
                        if I=0 then FATALERROR(3);
                        SEARCHT(K);
                        if K=0 then ERROR(0)
                        else with TPRED[K] do
                                begin PRECEDENCE:=I;ASSOCSYM:=TSYM
                                end;
                        INSYMBOL;
                    end;
                  TEST([PSYM]+S,S1,4)
              end;
        end;
procedure PRODUCTION(S,S1:SYMSET);
        label
            2;
            var L,K,I,NT1:integer;X:ALPHATYPE;TEMP:integer;
        begin INSYMBOL;
            repeat
                  NT1:=0;
                  if SY=IDSYM then SEARCHNT(NT1);
                  if NT1=0 then ERROR(3);
                  INSYMBOL;
                  if SY<> EQSYM then ERROR(7);
                  repeat MAXP:=MAXP+1;
                        if (MAXP>PRODMAX)then FATALERROR(4);
                        NTSONS[NT1]:=NTSONS[NT1]+1;TEMP:=NTARRAY[NT1];
```

```
                    NTARRAY[NT1]:=MAXP;with PRODARRAY[MAXP] do
                      begin NT:=NT1;NEXTPROD:=TEMP;PRODPTR:=PCOUNT+1
                      end;
                    INSYMBOL;L:=0;
                    while SY in [IDSYM,IDTSYM] do
                      begin SEARCHTNT(K,X);
                          if NUL then
                            begin
                                if L>0 then ERROR(0);
                                INSYMBOL;
                                L:=0;goto 2
                            end;
                          if K=0 then
                            begin L:=0;ERROR(0);INSYMBOL;goto 2
                            end;
                          PCOUNT:=PCOUNT+1;
                          if(PCOUNT>MAXPROD)then FATALERROR(5);
                          PROD[PCOUNT]:=X;L:=L+1;INSYMBOL
                      end;
                    2:
                    PRODARRAY[MAXP].LENGTH:=L;
                    PRODARRAY[MAXP].PRECEDENCE:=0;
                    if SY=PSYM then
                      begin INSYMBOL;
                          if (SY = IDSYM) or (SY = IDTSYM) then
                            begin SEARCHT(K);
                                if K=0 then ERROR(0)
                                else PRODARRAY[MAXP].PRECEDENCE:=TPRED[K].PRECEDEN
                            end;
                          INSYMBOL
                      end;
                    TEST([BARSYM,SEMICOLON],S+S1,9)
                until SY # BARSYM;
                if SY=SEMICOLON then INSYMBOL;
            until SY=ENDSYM;
        end;
begin(*MAIN READINPUT*);
    MAXP:=1;NTCOUNT:=1;TCOUNT:=3;
    PRCOUNT:=0; PCOUNT:=1;
    TSAVE[3]:='ERROR     ';NTSAVE[1]:='STARTP    ';
    NTARRAY[1]:=1;NTSONS[1]:=1;
    with PRODARRAY[1] do
      begin PRECEDENCE:=0;NT:=1;NEXTPROD:=0;
          PRODPTR:=1;LENGTH:=1
      end;
    with PROD[1] do
      begin ALPHA:=NONTERMINAL;NUM:=2
      end;
    for I:=2 to NTMAX do
      begin
          NTARRAY[I]:=0;NTSONS[I]:=0
```

```
        end;
      CH:=' ';CC:=0;LL:=0;ERRPOS:=0;ERRS:=[ ];
      TSAVE[1]:='NUL          ';TSAVE[2]:='S          ';INSYMBOL;
      FSYS:=[NTSYM,TSYM,PSYM,PRSYM,ENDSYM];
      if SY <> GRSYM then SKIP(FSYS,1)
      else INSYMBOL;
      TEST (FSYS,[],3);
      if SY=NTSYM then
          NONTERMS([TSYM],[PSYM,PRSYM,ENDSYM]);
      if SY=TSYM then
          TERMINALS([PSYM,PRSYM],[ENDSYM]);
      if SY=PSYM then
          PRECEDENCE([PRSYM],[ENDSYM]);
      if SY=PRSYM then
          PRODUCTION([ENDSYM],[ ]);
      if ERRS<>[ ] then
        begin ERRORMSG;WRITELN(ITY,'PROGRAM ABORTED');goto 1
        end;
  end;
procedure PRINT;
  var
      I:integer;
  begin WRITELN;
      for I:=1 to NTCOUNT do
          WRITELN(NTSAVE[I],NTSONS[I]);
      for I:=1 to TCOUNT do
          WRITELN(TSAVE[I]);
      for I:=1 to MAXP do
          with PRODARRAY[I] do
              WRITELN(NT,NEXTPROD,PRODPTR,LENGTH);
      for I:=1 to NTCOUNT do
          WRITELN('***',NTARRAY[I]);
  end;


procedure LAMBDA;
  var
      TEMP:packed array[1..NTMAX]of integer;
      LAM:packed array [0..NTMAX]of(YES,NO,UNDECIDED);
      TEMP1:packed array[1..PRODMAX]of boolean;
      I,NT1,L,PTR:integer;
      CHANGE,NOVER:boolean;
  begin for I:=1 to NTCOUNT do
        begin LAM[I]:=UNDECIDED;TEMP[I]:=NTSONS[I]
        end;
      for I:=1 to MAXP do TEMP1[I]:=false;
      for I:=1 to MAXP do
        begin with PRODARRAY[I] do
              begin NT1:=NT;L:=LENGTH;PTR:=PRODPTR
              end;
```

```
        if LAM[NT1]=UNDECIDED then
              if L=0 then LAM[NT1]:=YES
              else
                begin NOVER:=true;while (L > 0) and NOVER do
                      begin
                           if PROD[PTR].ALPHA=TERMINAL then
                             begin NOVER:=false;TEMP1[I]:=true;
                                 TEMP[NT1]:=TEMP[NT1]-1;
                                 if TEMP[NT1]=0 then LAM[NT1]:=NO
                             end
                           else
                             begin L:=L-1;PTR:=PTR+1
                             end;
                      end;
                end;
  end;
repeat CHANGE:=false;for I:=1 to MAXP do
              if not TEMP1[I] then
                begin with PRODARRAY[I] do
                      begin NT1:=NT;L:=LENGTH;PTR:=PRODPTR
                      end;
                    if LAM[NT1]=UNDECIDED then
                      begin NOVER:=true;while (L > 0)and NOVER do
                          case LAM[PROD[PTR].NUM] of
                                YES:
                                      if L=1 then
                                        begin LAM[NT1]:=YES;NOVER:=false;
                                            TEMP1[I]:=true;CHANGE:=true
                                        end
                                      else
                                        begin L:=L-1;PTR:=PTR+1
                                        end;
                                NO:
                                        begin NOVER:=false;TEMP1[I]:=true;
                                            TEMP[NT1]:=TEMP[NT1]-1;
                                            if TEMP[NT1]=0 then
                                              begin LAM[NT1]:=NO;CHANGE:=true
                                              end
                                        end;
                                UNDECIDED:
                                        begin L:=L-1;PTR:=PTR+1;
                                            while (L > 0) and (LAM[PROD[PTR].NUM] <
                                              begin L:=L-1;PTR:=PTR+1
                                              end;
                                        end
                          end;
                      end;
                end;
until not CHANGE;
CHANGE:=false;
for I:=1 to NTCOUNT do
```

```
                if LAM[I]=UNDECIDED then
                  begin WRITELN(I,' NT UNDECIDED');CHANGE:=true
                  end;
            if CHANGE then
              begin WRITELN(TTY,'NONTERMINAL UNDECIDED');goto 1
              end;
            for I:=1 to NTCOUNT do
                  if LAM[I]=YES then LAMDA[I]:=true
                  else LAMDA[I]:=false;
        end;

procedure ADDT(NUM:integer;var F:FIRSTSET);
        begin
            if NUM > 70 then F.H:=F.H+[NUM-70]
            else F.L:=F.L+[NUM]
        end;
procedure FIRST;
        type
            NTSET=record
                        H,M,L:set of 1..70
                  end;
        var
            FNT:array [1..NTMAX] of NTSET;
            NOVER:boolean;
            I,J,NT1,L,PTR:integer;
        procedure ADDNT(NUM:integer;var F:NTSET);
            begin
                if NUM > 140 then F.H:=F.H+[NUM-140]
                else
                        if NUM > 70 then F.M:=F.M+[NUM-70]
                        else F.L:=F.L+[NUM]
            end;
        function INSIDE(I:integer;F:NTSET):boolean;
            begin
                if I > 140 then INSIDE:=(I-140) in F.H
                else
                        if I > 70 then INSIDE:=(I-70) in F.M
                        else INSIDE:= I in F.L
            end;
        begin
            for I:=1 to NTCOUNT do
              begin FT[I].H:=[];FT[I].L:=[];
                  FNT[I].H:=[];FNT[I].M:=[];FNT[I].L:=[]
              end;
            for I := 1 to MAXP do
              begin NOVER:=true;
                  with PRODARRAY[I] do
                    begin
                        NT1:=NT;L:=LENGTH;PTR:=PRODPTR
                    end;
```

```
                while (L > 0) and NOVER do
                  begin
                      case PROD[PTR].ALPHA of
                          TERMINAL:
                                  begin ADDT(PROD[PTR].NUM,FT[NT1]);NOVER:=false
                                  end;
                          NONTERMINAL:
                                  begin ADDNT(PROD[PTR].NUM,FNT[NT1]);
                                      if LAMDA[NT1] then
                                        begin L:=L-1;PTR:=PTR+1
                                        end
                                      else NOVER:=false
                                  end
                      end;
                  end;
              end;
          for I := 1 to NTCOUNT do
                  for J := 1 to NTCOUNT do
                      if INSIDE(I,FNT[J]) then
                        begin FNT[J].H:=FNT[J].H+FNT[I].H;
                              FNT[J].M:=FNT[J].M+FNT[I].M;
                              FNT[J].L:=FNT[J].L+FNT[I].L;
                              FT[J].H:=FT[J].H+FT[I].H;
                              FT[J].L:=FT[J].L+FT[I].L
                        end;
      end;
procedure FPRINT;
      var
          I,J:integer;
      begin for I:= 1 to NTCOUNT do
            begin for J:= 1 to TCOUNT do
                      if J in FT[I].L then WRITE(TSAVE[J]);
                  if LAMDA[I] then WRITE(TSAVE[1]);
                  WRITELN;
            end;
      end;
procedure PROCITEMS;
      type
          X=array[1..TANDNTMAX]of record
                                      COUNT,NUM,KPTR:integer
                                  end;
          Y=array[1..TANDNTMAX]of integer;
      var
          KARRAY:array [1..MAXKERNELItem]of record
                                              NPTR,PRODNO,PLACE:integer
                                            end;
          TARRAY, NTARRAY1:X;
          TINDEX,NTINDEX:integer;
          STPTR,SNTPTR:Y;
          CPTR:integer;
          I:integer ;
```

```
procedure CLOSURE(NO:integer);
      var
          I,J,TIMES,PRODNO:integer;
      procedure ADD(CPRODNO,CPLACE:integer);
            var
                CLENGTH,CPRODPTR:integer;
            procedure INSERT(var C:X;var INDEX:integer;CNUM:integer);
                  var
                      I,P1,P2:integer;
                      NOVER:boolean;
                  begin with C[INDEX+1] do
                        begin NUM:=CNUM;KPTR:=0;COUNT:=0
                        end;
                      I:=1;while C[I].NUM <> CNUM do I:=I+1;
                      if I>INDEX then INDEX:=I;
                      P1:=0;P2:=C[I].KPTR;NOVER:=true;
                      while (P2 <> 0) and NOVER do with KARRAY[P2] do
                                  if(PRODNO<CPRODNO)or((PRODNO=CPRODNO)and(PLACE<CPL
                                  then
                                    begin P1:=P2;P2:=NPTR
                                    end
                                  else NOVER:=false;
                      if P1 <>0 then KARRAY[P1].NPTR:=CPTR
                      else C[I].KPTR:=CPTR;
                      KARRAY[CPTR].NPTR:=P2;
                      C[I].COUNT:=C[I].COUNT+1
                  end;
            begin
                with PRODARRAY[CPRODNO] do
                  begin CLENGTH:=LENGTH;CPRODPTR:=PRODPTR
                  end;
                if (CPLACE<CLENGTH)or(CLENGTH=0) then
                  begin CPTR:=CPTR+1;
                      if CPTR>MAXKERNELItems then FATALERROR(6);
                      with KARRAY[CPTR]do
                        begin PRODNO:=CPRODNO;PLACE:=CPLACE+1
                        end;
                      if CLENGTH <> 0 then
                            with PROD[CPRODPTR+CPLACE] do
                                  if ALPHA=TERMINAL then INSERT(TARRAY,TINDEX,NUM)
                                  else INSERT(NTARRAY1,NTINDEX,NUM)
                      else INSERT(TARRAY,TINDEX,1);
                  end;
            end;
      begin(*PROCEDURE CLOSURE*)
          TINDEX:=0;NTINDEX:=0;CPTR:=0;
          with STATES[NO] do
            begin TIMES:=NOOFITEMS;J:=KPTR
            end;
          for I:=1 to TIMES do
                with ITEMS[J] do
```

```
                    begin ADD(PRODNO,PLACE);J:=J+1
                    end;
              I:=1; while I<=NTINDEX do
                begin PRODNO:=NTARRAY[NTARRAY1[I],NUM];
                    while PRODNO<>0 do
                      begin ADD(PRODNO,0);PRODNO:=PRODARRAY[PRODNO].NEXTPROD
                      end;
                    I:=I+1
                end;
          end;
        (*CLOSURE*)
procedure NEXTSTATE;
        var
            I,J,FSTATE:integer;
            P:GOTOTYPE;P1:PGOTO;P2:NULPTR;
        procedure SEARCH(PTR1,LENGTH,NUM:integer;var FSTATE:integer;var C:Y);
            var
                I,PTR2:integer;NFOUND,NOVER:boolean;
            begin PTR2:=C[NUM];NFOUND:=true;
                while (PTR2 <> 0) and NFOUND do
                        with STATES[PTR2] do
                          begin
                              if LENGTH=NOOFITEMS then
                                begin I:=PTR1;J:=KPTR;NOVER:=true;
                                    while (I <> 0) and NOVER do
                                          if (KARRAY[I].PRODNO)#(ITEMS[J].PRODNO)
                                              then NOVER:=false
                                          else
                                              if (KARRAY[I].PLACE)=(ITEMS[J].PLACE)
                                                begin I:=KARRAY[I].NPTR;J:=J+1
                                                end
                                              else NOVER:=false;
                                    if NOVER then NFOUND:=false
                                    else PTR2:=NPTR
                                end
                              else PTR2:=NPTR
                          end;
                    if NFOUND then
                      begin STATECOUNT:=STATECOUNT+1;
                          if STATECOUNT > MAXSTATES then FATALERROR(7);
                          if (ITEMPTR+LENGTH) > MAXITEMS then FATALERROR(8);
                          FSTATE
                          :=STATECOUNT;
                          with STATES[STATECOUNT] do
                            begin NOOFITEMS:=LENGTH;KPTR:=ITEMPTR+1;
                              NPTR:=C[NUM];PGOTO:=nil;NULPROD:=nil
                            end;
                          C[NUM]:=STATECOUNT;
                          while (PTR1 <> 0) do
                            begin ITEMPTR:=ITEMPTR+1;
                                with KARRAY[PTR1] do
```

```
                                    begin ITEMS[ITEMPTR].PRODNO:=PRODNO;
                                        ITEMS[ITEMPTR].PLACE:=PLACE;
                                        PTR1:=NPTR
                                    end;
                                end;
                            end
                        else FSTATE:=PTR2
                    end;
            begin (*NEXTSTATE*)
                I:=1;
                while I <= TINDEX do
                  begin with TARRAY[I] do
                        if NUM <> 1 then
                          begin SEARCH(KPTR,COUNT,NUM,FSTATE,STPTR);
                            NEW(P);with P^ do
                              begin NEXTSTATE:=FSTATE;
                                ACTION:=SHIFT;
                                SYMBOL:=NUM;
                                PTRN:=STATES[CSTATE].PGOTO
                              end;
                            STATES[CSTATE].PGOTO:=P
                          end
                        else
                          begin J:=KPTR;
                            while J <> 0 do
                              begin NEW(P2);with P2^ do
                                begin PRODNO:=KARRAY[J].PRODNO;
                                    NEXTNUL:=STATES[CSTATE].NULPROD
                                end;
                                STATES[CSTATE].NULPROD:=P2;
                                J:=KARRAY[J].NPTR
                              end;
                          end;
                    I:=I+1;
                  end;
                I:=1;
                while I <= NTINDEX do
                  begin with NTARRAY1[I] do
                        SEARCH(KPTR,COUNT,NUM,FSTATE,SNTPTR);
                    NEW(P1);with P1^ do
                      begin CURRENTSTATE:=CSTATE;
                        NEXTSTATE:=FSTATE;
                        PTR:=TGOTO[NTARRAY1[I].NUM]
                      end;
                    TGOTO[NTARRAY1[I].NUM]:=P1;I:=I+1
                  end;
            end;
        procedure PRINTCLOSURE(CS:integer);
            var
                I:integer;
            begin WRITELN('**** CLOSURE FOR STATE ',CS:3);
```

```
                for I:=1 to CPTR do with KARRAY[I] do
                        WRITELN(NPTR,PRODNO,PLACE);
            end;
        procedure PRINTSTATES;
            var
                I:integer;P:GOTOTYPE;P1:NULPTR;P2:BGOTO;
            begin for I:=1 to STATECOUNT do
                    with STATES[I] do
                      begin WRITELN('**** STATE',I:3);
                            WRITELN(NOOFITEMS,KPTR,NPTR);P:=PGOTO;
                            while P <> nil do with P^ do
                                    begin WRITELN(SYMBOL,NEXTSTATE);P:=PTR
                                    end;
                            P1:=NULPROD;while P1 <> nil do with P1^ do
                                    begin WRITELN('****NUL PRODUCTIONS',PRODNO);P1:=NEXT
                                    end;
                      end;
                for I:= 1 to NTCOUNT do
                  begin P2:=TGOTO[I];
                        while P2 <> nil  do with P2^ do
                          begin WRITELN('GOTO',CURRENTSTATE,NEXTSTATE);P2:=PTR
                          end;
                  end;
                WRITELN('****DUMP OF ITEMS');
                for I :=1 to ITEMPTR do with ITEMS[I] do
                        WRITELN(I,PRODNO,PLACE)
            end;
        begin (*PROCITEMS*)
            CSTATE:=1;STATECOUNT:=1;ITEMPTR:=1;
            with STATES[1] do
              begin NOOFITEMS:=1;KPTR:=1;CPTR:=1;NPTR:=0;PGOTO:=nil;NULPROD:=nil;
              end;
            with ITEMS[1] do
              begin PRODNO:=1;PLACE:=0
              end;
            for I:= 1 to NTMAX do
              begin SIPTR[I]:=0;SNTPTR[I]:=0;TGOTO[I]:=nil
              end;
            while CSTATE <= STATECOUNT do
              begin CLOSURE(CSTATE);
                    (*$PRINTCLOSURe(CSTATE);$*)
                    NEXTSTATE;
                    CSTATE:=CSTATE+1
              end;
            (*$PRINTSTATEs;$*)
        end;

    procedure ITEMINTIALise;
        var
            I:integer;
        function LAMDADO(PRODNO,PLACE:integer):boolean;
```

```
            label
                2;
                var PTR,LEN:integer;
            begin LAMDADO:=false;
                with PRODARRAY[PRODNO] do
                  begin PTR:=PRODPTR+PLACE; LEN:=LENGTH
                  end;
                if PROD[PTR-1].ALPHA=NONTERMINAL then
                  begin
                      while PLACE < LEN do
                            if PROD[PTR].ALPHA=TERMINAL then goto 2
                            else
                                  if LAMDA[PROD[PTR].NUM] then
                                    begin PLACE:=PLACE+1; PTR:=PTR+1
                                    end
                                  else goto 2;
                      LAMDADO:=true;
                  end;
                2:
            end;
      begin
          I:=2;
          with ITEMS[1] do
            begin LADONE:=true;PREDPTR:=nil;LAM:=true
            end;
          while I <= ITEMPTR do
                with ITEMS[I] do
                  begin LADONE:=false;
                      PREDPTR:=nil;
                      LAM:=LAMDADO(PRODNO,PLACE);
                      I:=I+1
                  end;
      end
          ;
procedure GOTOT(I:integer;var NSTATE:integer);
      var
          P:GOTOTYPE;
      begin P:=STATES[NSTATE].PGOTO;
          while (P^.SYMBOL <> I ) do P:=P^.PTRN;
          NSTATE:=P^.NEXTSTATE
      end;
procedure GOTONT(I:integer;var NSTATE:integer);
      var
          P:BGOTO;
      begin P:=TGOTO[I];
          while (P^.CURRENTSTAte <> NSTATE) do P:=P^.PTR;
          NSTATE:=P^.NEXTSTATE
      end;
procedure CPREDSTATE;
      var
          I:integer;PTR1:BGOTO;
```

```
        procedure PRODDO(PRODNO,STATE:integer);
              var
                  I,PTR,PTR1,NSTATE,LEN:integer;
                  P:PREPTR;PPTR,PSTATE:integer;
              begin with PRODARRAY[PRODNO] do
                    begin PTR:=PRODPTR;LEN:=LENGTH
                    end;
                  I:=1;NSTATE:=STATE;PTR1:=0;
                  while I <= LEN do
                    begin PSTATE:=NSTATE;PPTR:=PTR1;
                        if PROD[PTR].ALPHA=TERMINAL
                              then GOTOT(PROD[PTR].NUM,NSTATE)
                        else GOTONT(PROD[PTR].NUM,NSTATE);
                        PTR1:=STATES[NSTATE].KPTR;
                        while not ((ITEMS[PTR1].PRODNO = PRODNO) and
                        (ITEMS[PTR1].PLACE = I)) do
                              PTR1:=PTR1+1;
                        if I = LEN then
                          begin NEW(P);P^.NPTR:=ITEMS[PTR1].PREDPTR;
                              P^.STATENO:=STATE;ITEMS[PTR1].PREDPTR:=P
                          end;
                        if ITEMS[PTR1].LAM then
                              if (PPTR # 0) then
                                begin NEW(P);P^.NPTR:=ITEMS[PPTR].PREDPTR;
                                    P^.STATENO:=STATE;ITEMS[PPTR].PREDPTR:=P
                                end;
                        I:=I+1 ;PTR:=PTR+1;
                    end;
              end;
        procedure NTDO(NT,STATE:integer);
              var
                  PRODNO:integer;
              begin PRODNO:=NTARRAY[NT];
                  while PRODNO # 0 do
                    begin PRODDO(PRODNO,STATE);
                        PRODNO:=PRODARRAY[PRODNO].NEXTPROD
                    end;
              end;
        begin NTDO(1,1);for I:=2 to NTCOUNT do
              begin PTR1:=TGOTO[I];
                  while PTR1 # nil do
                    begin NTDO(I,PTR1^.CURRENTSTATE);
                        PTR1:=PTR1^.PTR
                    end;
              end;
        end;
    procedure LALRSET(PRODNO,PLACE,STATE:integer;var LA:LASET);
        var
            DARRAY:array[1..DONEMAX] of record
                                            NT,STATENO:integer
                                        end;
```

```
    PTRDONE:integer;
procedure FIRSTB(PRODNO,PLACE:integer);
      label
          2;
          var J,PTR,LEN:integer;
      begin with PRODARRAY[PRODNO] do
            begin PTR:=PRODPTR+PLACE;LEN:=LENGTH
            end;
          while PLACE < LEN do
            begin J:=PROD[PTR].NUM;
                if PROD[PTR].ALPHA=TERMINAL then
                  begin ADDI(J,LA);goto 2
                  end
                else
                  begin LA.H:=LA.H+FT[J].H;
                        LA.L:=LA.L+FT[J].L
                  end;
                if LAMDA[J] then
                  begin PLACE:=PLACE+1;PTR:=PTR+1
                  end
                else goto 2
            end;
          2;
      end;
procedure STATEFIRST(STATENO:integer);
      var
          TIMES,I,PTR:integer;
      begin with STATES[STATENO] do
            begin TIMES:=NOOFITEMS;PTR:=KPTR
            end;
          for I:=1 to TIMES do
                with ITEMS[PTR] do
                  begin FIRSTB(PRODNO,PLACE);PTR:=PTR+1
                  end;
      end;
function DONE(NTNUM,STATE:integer):boolean;
      var
          I,J:integer;
      begin J:=PTRDONE+1;I:=1;
          if J > DONEMAX then FATALERROR(9);
          with DARRAY[J] do
            begin NT:=NTNUM;STATENO:=STATE
            end;
          while not((DARRAY[I].NT=NTNUM)and
          (DARRAY[I].STATENO=STATE)) do
                I:=I+1;
          if I=J then
            begin PTRDONE:=I;DONE:=false
            end
          else DONE:=true
      end;
```

```
procedure LALRP(PRODNUM,PLACENO,STATENO:integer);
      label
          2;
          var PTR1:PREPTR;
          PREDS,PTRK,NT:integer;
          TIMES,TPTR,TEMP,I:integer;
      begin
          if PLACENO=0 then
            begin PREDS:=STATENO;PTR1:=nil
            end
          else
            begin PTRK:=STATES[STATENO].KPTR;
                while not ((ITEMS[PTRK].PRODNO=PRODNUM) and
                (ITEMS[PTRK].PLACE=PLACENO)) do
                      PTRK:=PTRK+1;
                if ITEMS[PTRK].LADONE then
                  begin with ITEMS[PTRK] do
                        begin LA.H:=LALR.H+LA.H;
                              LA.L:=LALR.L+LA.L
                        end;
                      goto 2
                  end;
                PTR1:=ITEMS[PTRK].PREDPTR;
            end;
          NT:=PRODARRAY[PRODNUM].NT;
          repeat
                if NT=1 then
                  begin LA.L:=LA.L+[2];goto 2;
                  end;
                if PTR1 <> nil then  with PTR1^ do
                        begin PREDS:=STATENO;PTR1:=NPTR
                        end;
                if not DONE(NT,PREDS) then
                  begin TEMP:=PREDS;GOTONT(NT,TEMP);
                      STATEFIRST(TEMP);
                      with STATES[TEMP] do
                        begin
                            TIMES:=NOOFITEMS;TPTR:=KPTR
                        end;
                      for I:=1 to TIMES do
                        begin with ITEMS[TPTR] do
                                if LAM then LALRP(PRODNO,PLACE-1,PREDS);
                            TPTR:=TPTR+1
                        end;
                  end;
          until
                (PTR1=nil);
          2:
      end;
begin PTRDONE:=0;LA.H:=[];LA.L:=[ ];
    if PRODARRAY[PRODNO].NT=1 then LA.L:=LA.L+[2]
```

```
            else LALRP(PRODNO,PLACE,STATE);
        end;
procedure LALRCOMPUTE;
        var
            LA:LASET;
            TIMES,I,J,PTR:integer;
            X:NULPTR;
        begin
            with ITEMS[1] do
              begin LADONE:=true;LALR.H:=[];LALR.L:=[2];
              end;
            for I:=1 to STATECOUNT do
              begin with STATES[I] do
                    begin TIMES:=NOOFITEMS;PTR:=KPTR
                    end;
                for J:=1 to TIMES do
                  begin    with ITEMS[PTR] do
                             if PREDPTR <> nil then
                               begin LALRSET(PRODNO,PLACE,I,L
                                 LADONE:=true;LALR:=LA;
                               end;
                       PTR:=PTR+1
                  end;
                X:=STATES[I].NULPROD;
                while X <> nil do
                  begin LALRSET(X^.PRODNO,0,I,LA);
                        X^.LALR:=LA;X:=X^.NEXTNUL
                  end;
              end;
        end;
procedure PRINTLALR;
        var
            TIMES,I,J,PTRK:integer;
            P:NULPTR;
        procedure PREDPRINT(P:PREPTR);
            var
                J:integer;
            begin WRITE('PREDS ');J:=0;
                while P <> nil do
                  begin J:=J+1;
                      if J > 15 then
                        begin J:=1;WRITELN;WRITE('
                        end;
                      WRITE(P^.STATENO:5);
                      P:=P^.NPTR
                  end;
                WRITELN;
            end;
        procedure LAPRINT(LA:LASET);
            var
                I,J:integer;PRB:boolean;
```

```
            begin WRITE('          LALR  ');J:=0;
                for I:=1 to TCOUNT do
                  begin PRB:=false;
                      if I <= 70 then
                          if I in LA.L then PRB:=true;
                      if I > 70 then
                          if (I-70) in LA.H then PRB:=true;
                      if PRB then
                        begin J:=J+1;
                            if J > 7 then
                              begin J:=1;WRITELN;WRITE('
                              end;
                            WRITE(TSAVE[I]);
                        end;
                  end;
                WRITELN;
            end;
      begin for I:=1 to STATECOUNT do
            begin with STATES[I] do
                  begin TIMES:=NODEITEMS;PTRK:=KPTR;P:=NILPROD
                  end;
            WRITELN;WRITELN('****STATE NO ',I:3);
            for J:=1 to TIMES do
              begin with ITEMS[PTRK] do WRITE('[',PRODNO:3,',',PLACE:3,']');
                  if ITEMS[PTRK].LADONE then
                    begin PREDPRINT(ITEMS[PTRK],PREDPTR);
                      LAPRINT(ITEMS[PTRK],LALR)
                    end;
                  PTRK:=PTRK+1;WRITELN;
              end;
                while P <> nil do
                  with P^ do
                    begin WRITELN('[',PRODNO:3,',  0]');
                      LAPRINT(LALR);P:=P^.NEXTNUL
                    end;
            end;
      end;
procedure INITPARSE;
      begin WRITELN(PARSEFILE,'END;');
          WRITELN(PARSEFILE);
          WRITELN(PARSEFILE,'INITPROCEDURE;');
          WRITELN(PARSEFILE,'BEGIN');
          INITCOUNT:=0;
      end;
procedure OUTPARSETABLE;
      type
          ACTIONTYPE=(SHIFT,REDUCE,NOACTION);
      var
          PARRAY:array[0..TMAX] of record
                                     TNO,STATEPROD:integer;
                                     ACTION:ACTIONTYPE
```

```
            begin WRITE('          LALR  ');J:=0;
                for I:=1 to TCOUNT do
                  begin PRB:=false;
                      if I <= 70 then
                          if I in LA.L then PRB:=true;
                      if I > 70 then
                          if (I-70) in LA.H then PRB:=true;
                      if PRB then
                        begin J:=J+1;
                            if J > 7 then
                              begin J:=1;WRITELN;WRITE('
                              end;
                            WRITE(TSAVE[I]);
                        end;
                  end;
                WRITELN;
            end;
      begin for I:=1 to STATECOUNT do
            begin with STATES[I] do
                begin TIMES:=NODEITEMS;PTRK:=KPTR;P:=NULPROD
                end;
            WRITELN;WRITELN('****STATE NO ',I:3);
            for J:=1 to TIMES do
              begin with ITEMS[PTRK] do WRITE('[',PRODNO:3,',',PLACE:3,']');
                  if ITEMS[PTRK].LADONE then
                    begin PREDPRINT(ITEMS[PTRK],PREDPTR);
                      LAPRINT(ITEMS[PTRK].LALR)
                    end;
                  PTRK:=PTRK+1;WRITELN;
              end;
            while P <> nil do
                  with P^ do
                    begin WRITELN('[',PRODNO:3,',  01');
                      LAPRINT(LALR);P:=P^.NEXTNUL
                    end;
            end;
      end;
procedure INITPARSE;
      begin WRITELN(PARSEFILE,'END;');
          WRITELN(PARSEFILE);
          WRITELN(PARSEFILE,'INITPROCEDURE;');
          WRITELN(PARSEFILE,'BEGIN');
          INITCOUNT:=0;
      end;
procedure OUTPARSETABLE;
      type
          ACTIONTYPE=(SHIFT,REDUCE,NOACTION);
      var
          PARRAY:array[0..TMAX] of record
                                     TNO,STATEPROD:integer;
                                     ACTION:ACTIONTYPE
```

```
                                        end;
    X:GOTOTYPE;Y:NULPTR;
    I,PTR,MAY,MAXPROD,MAXITEMS:integer;
    ERRORNO,KPTR,ITEMCOUNT,SAVED:integer;
    STATEHEAD,SRERRORS,ERRORV:boolean;
procedure  RESOLVECONflict(J,PRODNO:integer;var ACTIONS:ACTIONTYPE);
      var
          CONFLICT:(SR,RR);
          TNUM,PR1,PR2:integer;RESOLVE:boolean;
      begin with  PARRAY[J] do
          begin TNUM:=TNO;
              if ACTION = SHIFT then
                begin CONFLICT:=SR;PR1:=TPRED[TNUM].PRECEDENCE
                end
              else
                begin CONFLICT:=RR;PR1:=PRODARRAY[STATEPROD].PRECEDENCE
                end;
          end;
          RESOLVE:=false;ACTIONS:=NOACTION;
          PR2:=PRODARRAY[PRODNO].PRECEDENCE;
          if TNUM = 3 then RESOLVE:=true;
          if (PR1 <> 0 )and(PR2 <> 0) then
              if PR1 < PR2 then
                begin ACTIONS:=REDUCE;RESOLVE:=true
                end
              else
                    if PR1 > PR2 then RESOLVE:=true
                    else
                          if CONFLICT=SR then
                          case TPRED[TNUM].ASSOCSYM of
                                LSYM:
                                      begin ACTIONS:=REDUCE;RESOLVE:=true
                                      end;
                                RSYM:RESOLVE:=true
                          end;
          if not RESOLVE then
            begin SRERRORS:=true;
              if STATEHEAD then
                begin WRITELN;WRITELN('*****ERRORS IN STATE=',CSTATE:4);
                      STATEHEAD:=false
                end;
              case CONFLICT of
                    SR:WRITELN('  SHIFT REDUCE ERROR,TNO=',TNUM:3,'PROD NO=',PRODN
                    RR:WRITELN('  REDUCE REDUCE ERROR,TNO=',TNUM:3,'PRODS=',
                               PRODNO:4,PARRAY[J].STATEPROD:5)
              end;
            end;
      end;
procedure PUTPROD(PRODNO:integer;LALR:LASET);
      var
          NOOFSYMBOLS,I,J:integer;
```

```
        NFOUND:boolean;
        ACTIONS:ACTIONTYPE;
      function INSIDE(I:integer):boolean;
        begin
          if I <= 70 then INSIDE:= I in LALR.L
          else INSIDE:=(I - 70 ) in LALR.H
        end;
      begin NOOFSYMBOLS:=0;
        for I:=1 to ICOUNT do
          if INSIDE(I) then
            begin NFOUND:=true;J:=1;
              while (J <= PTR) and NFOUND do
                if PARRAY[J].IND=I then NFOUND:=false
                else J:=J+1;
              if NFOUND then
                begin PTR:=PTR+1;with PARRAY[PTR] do
                  begin IND:=I;STATEPROD:=PRODNO;
                    ACTION:=REDUCE;NOOFSYMBOLS:=NOOFSYMBOLS+1
                  end
                end
              else
                begin RESOLVECONflict(J,PRODNO,ACTIONS);
                  if ACTIONS # NOACTION then  with PARRAY[J] do
                    begin ACTION:=REDUCE;STATEPROD:=PRODNO;
                      NOOFSYMBOLS:=NOOFSYMBOLS+1;
                    end;
                end;
            end;
        if NOOFSYMBOLS > MAX then
          begin MAX:=NOOFSYMBOLS;MAXPROD:=PRODNO
          end;
      end;
    procedure OUTPUTSTATE;
      var
        COUNT,SAVE,I:integer;TEMP:integer;
      procedure OUT1;
        var
          PTR,I:integer;
        begin PTR:=SAVE -1;
          for I:=1 to COUNT do
            begin PTR:=PTR+1;
              TEMP:=TEMP+1;
              if TEMP=3 then
                begin WRITELN(PARSEFILE);TEMP:=1
                end;
              if TPARSE[PTR].ACTION=SH  then WRITE(PARSEFILE,'PT[',PTR:4,'].A:=R ;');
              else WRITE(PARSEFILE,'PT[',PTR:4,'].A:=R ;');
              WRITE(PARSEFILE,'PT[',PTR:4,'].I:=',TPARSE[PTR].T:3,';');
              WRITE(PARSEFILE,'PT[',PTR:4,'].NUM:=',TPARSE[PTR].NUM:4,';');
            end;
          INITCOUNT:=INITCOUNT+COUNT;
```

```
            if INITCOUNT > 50 then
              begin WRITELN(PARSEFILE);WRITELN(PARSEFILE,'END;');
                  WRITELN(PARSEFILE,'INITPROCEDURE;');
                  WRITELN(PARSEFILE,'BEGIN');INITCOUNT:=0
              end;
        end;
procedure OUT(I:integer;J:boolean);
        begin MAXITEMS:=MAXITEMS+1;COUNT:=COUNT+1;
            if MAXITEMS > MAXTSIZE then
              begin WRITELN(TTY,'MAXTSIZE EXCEEDED');goto 1
              end;
            with TPARSE[MAXITEMS] do
              begin
                  if J then ACTION:=SH
                  else ACTION:=RE;
                  T:=PARRAY[I].TNO; NUM:=PARRAY[I].STATEPROD
              end;
        end;
procedure COMPARE;
        var
            NOVER,FOUND:boolean;
            PTR1,CURRENT,I,J:integer;
        begin NOVER:=true;CURRENT:=TEMP1[COUNT];FOUND:=false;
            if COUNT > 1 then
                  while (CURRENT # 0) and NOVER do
                    begin FOUND:=true;T:=SAVE;J:=1;PTR1:=TEMP2[CURRENT].PTR;
                        while FOUND and (J <= COUNT ) do
                              if (TPARSE[I].ACTION=TPARSE[PTR1].ACTION ) and
                              (TPARSE[I].T=TPARSE[PTR1].T) and
                              (TPARSE[I].NUM=TPARSE[PTR1].NUM) then
                                begin I:=I+1;J:=J+1;PTR1:=PTR1+1
                                end
                              else FOUND:=false;
                        if FOUND then NOVER:=false
                        else CURRENT:=TEMP2[CURRENT].NEXT;
                    end;
            if FOUND then
              begin MAXITEMS:=SAVE-1;SAVE:=TEMP2[CURRENT].PTR;
                  MAX:=COUNT+1
              end
            else
              begin OUT1;TEMP2PTR:=TEMP2PTR+1;
                  with TEMP2[TEMP2PTR] do
                    begin NEXT:=TEMP1[COUNT];PTR:=SAVE
                    end;
                  TEMP1[COUNT]:=TEMP2PTR
              end;
        end;
begin WRITELN(PARSEFILE);COUNT:=0;SAVE:=MAXITEMS+1;TEMP:=0;
    with PARRAY[0] do
      begin TNO:=0;STATEPROD:=MAXPROD
```

```
            end;
          if ERRORV then
            begin OUT(ERRORNO,true);MAXPROD:=0;MAX:=1
            end
          else
                if MAX < 3 then
                  begin MAXPROD:=0;MAX:=1
                  end;
          for  I := 1 to PTR do with PARRAY[I] do
                      if ACTION=SHIFT then
                        begin
                            if RNO # 3 then OUT(I,true)
                        end
                      else
                              if STATEPROD # MAXPROD then OUT(I,false);
          if MAXPROD # 0 then OUT(0,false);
          COMPARE;
          WRITELN(PARSEFILE);
          WRITE(PARSEFILE,'STATES[',CSTATE:4,'],PTR:=',SAVE:5,';');
          WRITELN(PARSEFILE,'STATES[',CSTATE:4,'].ENTRIES:=',COUNT:3,';');
          SAVED:=SAVED+MAX-1
      end;
procedure WRITEGOTO;
      var
          I,J,PTR,SPTR,COUNT:integer;
          X:BGOTO;
      begin WRITELN(PARSEFILE);PTR:=0;INITPARSE;
          for I :=1 to NTCOUNT do
            begin SPTR:=PTR+1;COUNT:=0;J:=0;X:=TGOTO[I];
                while X # nil do
                  begin COUNT:=COUNT+1;PTR:=PTR+1;J:=J+1;
                      if J = 3 then
                        begin J:=0;WRITELN(PARSEFILE)
                        end;
                      WRITE(PARSEFILE,'TGOTO[',PTR:4,'].CS:=',X^.CURRENTSTATE:4,';');
                      WRITE(PARSEFILE,'TGOTO[',PTR:4,'].NS:=',X^.NEXTSTATE:4,';');
                      X:=X^.PTR
                  end;
                WRITELN(PARSEFILE);
                WRITE(PARSEFILE,'NTGOTO[',I:3,'],PTR:=',SPTR:4,';');
                WRITELN(PARSEFILE,'NTGOTO[',I:3,'].NO:=',COUNT:4,';');
                WRITELN(PARSEFILE);
                INITCOUNT:=INITCOUNT+COUNT;
                if INITCOUNT > 50 then INITPARSE;
            end;
          WRITELN('GOMAX=',PTR)
      end;
begin
    MAXITEMS:=0;CSTATE:=1;SAVED:=0;SRERRORS:=false;
    INITCOUNT:=0;TEMP2PTR:=0;
    for I:= 1 to TMAX do TEMP1[I]:=0;
```

```
            while CSTATE <= STATECOUNT do
              begin MAX:=0;ERRORV:=false;PTR:=0;ERRORNO:=0;
                  STATEHEAD:=true;X:=STATES[CSTATE].PGOTO;
                  while X <> nil do
                    begin PTR:=PTR+1; with PARRAY[PTR] do
                          begin TNO:=X^.SYMBOL;STATEPROD:=X^.NEXTSTATE;ACTION:=SHIFT;
                              if TNO=3 then
                                begin ERRORV:=true;ERRORNO:=PTR
                                end;
                          end;
                        X:=X^.PTRN
                    end;
                  ITEMCOUNT:=STATES[CSTATE].NOOFITEMS;I:=1;
                  KPTR:=STATES[CSTATE].KPTR;
                  while I <= ITEMCOUNT do
                    begin with ITEMS[KPTR] do
                              if PRODARRAY[PRODNO].LENGTH=PLACE then PUTPROD(PRODNO,LALR);
                        I:=I+1;KPTR:=KPTR+1
                    end;
                  Y:=STATES[CSTATE].NULPROD;
                  while Y <> nil do with Y^ do
                              begin PUTPROD(PRODNO,LALR);Y:=NEXTNUL
                              end;
                  OUTPUTSTATE;CSTATE:=CSTATE+1
              end;
            WRITEGOTO;
            if SRERRORS then WRITELN(TTY,'SHIFT REDUCE ERRORS');
            WRITELN(TTY,'MAXITEMS=',MAXITEMS,'SAVED=',SAVED);
            WRITELN('MAXITEMS=',MAXITEMS,'SAVED=',SAVED);
        end;
    procedure
    DUMPALL;
        procedure WRITETERMS;
            var
                I:integer;
            begin WRITELN;WRITELN('*****TERMINALS');
                for I:=1 to TCOUNT do
                  begin WRITE('    ',I:3,'  ',TSAVE[I],TPRED[I].PRECEDENCE:3,' ');
                      case TPRED[I].ASSOCSYM of
                              LSYM:WRITE('LEFT');
                              RSYM:WRITE('RIGHT');
                              NONSYM:WRITE('NONASSOC')
                      end;
                      WRITELN;
                  end;
                WRITELN;
            end;
        procedure WRITENONTErms;
            var
                I:integer;
            begin WRITELN;WRITELN('*****NONTERMINALS');;
```

```
                for I:= 1 to NTCOUNT do
                    WRITELN('     ',I:3,'  ',NTSAVE[I]);
                WRITELN;
            end;
        procedure WRITEPROD;
            var
                I,J,L,COUNT:integer;
            begin WRITELN;WRITELN('*****PRODUCTIONS');
                for I:=1 to MAXP do
                  begin WRITELN;with PRODARRAY[I] do
                        begin WRITE('    ',I:3,PRECEDENCE:3,'  ',NTSAVE[NT],'=');
                            J:=PRODPTR;L:=LENGTH
                        end;
                    COUNT:=0;
                    if L=0 then WRITE(TSAVE[1])
                    else while L > 0 do
                            begin COUNT:=COUNT+1;
                                if COUNT=7 then
                                  begin COUNT:=0;WRITELN;WRITE(' ':20)
                                  end;
                                with PROD[J] do
                                    if ALPHA=TERMINAL then WRITE(TSAVE[NUM])
                                    else WRITE(NTSAVE[NUM]);
                                J:=J+1;L:=L-1
                            end;
                    WRITELN(';');
                  end;
            end;
        begin WRITELN;
            WRITELN('   NTMAX=',NTCOUNT,'   TMAX=',TCOUNT);
            WRITELN('   MAXP =',MAXP,'   MAXSTATE=',STATECOUNT);
            WRITETERMS;WRITENONTErms;WRITEPROD
        end;
    procedure PRODDUMP;
        var
            I,COUNT:integer;
        begin COUNT:=0;WRITELN(PARSEFILE);INITPARSE;
            for I:=1 to MAXP do
              begin COUNT:=COUNT+1;
                    if COUNT = 4 then
                      begin WRITELN(PARSEFILE);COUNT:=1
                      end;
                    WRITE(PARSEFILE,'PRA[',I:3,'].NT:=',PRODARRAY[I].NT:3);
                    WRITE(PARSEFILE,';  PRA[',I:3,'].LEN:=',PRODARRAY[I].LENGTH:2,';  ');
              end;
            WRITELN(PARSEFILE);
        end;
    procedure TDUMP;
        var
            I,COUNT:integer;
        begin COUNT:=0;WRITELN(PARSEFILE);INITPARSE;
```

```
            for I:=1 to TCOUNT do
              begin COUNT:=COUNT+1;
                  if COUNT=5 then
                    begin WRITELN(PARSEFILE);COUNT:=0
                    end;
                  WRITE(PARSEFILE,'IT(',I:3,'):=''',TSAVE[I],''';');
              end;
            WRITELN(PARSEFILE);
        end;
begin READINPUT;LAMBDA;FIRST;PROCITEMS;
    ITEMINTIALise;CPREDSTATE;LALRCOMPUTe;PRINTLALR;
    PARSENAME:='PARSET    ';
    REWRITE(PARSEFILE,PARSENAME);WRITELN(PARSEFILE);
    WRTTELN(PARSEFILE,'INITPROCEDURE;');WRITELN(PARSEFILE,'BEGIN');
    OUTPARSETAble;PRODDUMP;TDUMP;WRITELN(PARSEFILE,'END;');
    DUMPALL;
    1:
end.
```

```
PROGRAM PARSER(INPUT,OUTPUT);
  label
      1;
      (* THESE CONSTANTS ARE GIVEN BY THE LALR PROGRAM *)
      const NTMAX=          ;TMAX=          ;MAXP=          ;MAXSTATES=
      MAXITEMS=         ;GOMAX=          ;STKSIZE=100;
  type
      SYMTYPE=(TERMINAL,NONT,OTHERSSY);
      STACKCONTEnts=record
                              STYPE:SYMTYPE;
                              STATE,TNO:integer
                    end;
      ACTIONTYPE=(S,R,ERROR);
      ALPHA=packed array [1 .. 10] of char;
  var
      STATES :array [1 .. MAXSTATES] of record
                                          PTR,ENTRIES:integer
                                        end;
      PT:array [1 .. MAXITEMS] of record
                                    A:ACTIONTYPE;
                                    T,NUM:integer
                                  end;
      NTGOTO:array [1 .. NTMAX] of record
                                     PTR,NO:integer
                                   end;
      TGOTO:array[1 .. GOMAX] of record
                                    CS,NS:integer
                                  end;
      PRA:array[1 .. MAXP] of record
                                NT,LEN :integer
                              end;
      TT:array[1 .. TMAX] of ALPHA;
      STACK:array [0 .. STKSIZE] of STACKCONTEnts;
      SCSYMBOL,CSYMBOL,CSTATE:integer;ERRORSET,PARSEOVER:boolean;
      STKPTR:integer;
      ID:ALPHA;
      CH:char;EOL:boolean;
      CHCNT:integer;

  procedure INSYMBOL(var CSYMBOL:integer);
      forward;
  procedure ERRORRECOVery;
      begin
      end;
  procedure GETACTION(var CACTION:ACTIONTYPE;var NEXT:integer);
      var
          I,PTR1,TIMES:integer;NFOUND:boolean;
      begin I:=1;CACTION:=ERROR;NFOUND:=true;
          with STATES[CSTATE] do
            begin PTR1:=PTR;TIMES:=ENTRIES
            end;
          while (I <= TIMES) and NFOUND do
                with PT[PTR1] do
```

```
                        if (CSYMBOL = T ) or (T=0) then
                          begin CACTION:=A;NEXT:=NUM;NFOUND:=false
                          end
                        else
                          begin I:=I+1;PTR1:=PTR1+1
                          end;
        end;
procedure PARSERUN;
        var
            CACTION:ACTIONTYPE;TIMES,NTNUM,NEXT,PTR1:integer;
            NOEXIT:boolean;
        begin ERRORSET:=false;NOEXIT:=true;GETACTION(CACTION,NEXT);
            case CACTION of
                    S:
                        begin STKPTR:=STKPTR+1;
                            if STKPTR > STKSIZE then
                              begin WRITELN(TTY,'STACKSIZE EXCEEDED');goto 1
                              end;
                            with STACK[STKPTR] do
                              begin TNO:=CSYMBOL;STATE:=CSTATE;STYPE:=TERMINAL
                              end;
                            CSTATE:=NEXT;INSYMBOL(CSYMBOL)
                        end;
                    R:
                        begin
                            if NEXT=1 then PARSEOVER:=true
                            else with PRA[NEXT] do
                                        if LEN <> 0 then
                                          begin STKPTR:=STKPTR-LEN;CSTATE:=STACK[STKPTR+1].S
                                          end;
                            if not PARSEOVER then
                              begin NTNUM:=PRA[NEXT].NT;STKPTR:=STKPTR+1;with STACK[STKPTR]
                                    begin STYPE:=NONT;STATE:=CSTATE;TNO:=NTNUM
                                    end;
                                PTR1:=NTGOTO[NTNUM].PTR;TIMES:=NTGOTO[NTNUM].NO;
                                while TGOTO[PTR1].CS <> CSTATE do PTR1:=PTR1+1;
                                CSTATE:=TGOTO[PTR1].NS
                              end;
                        end;
                ERROR:ERRORSET:=true
            end;
        end;
begin STKPTR:=0;CSTATE:=1;PARSEOVER:=false;
    CHCNT:=0;CH:=' ';CSYMBOL:=0;
    EOL:=false;INSYMBOL(CSYMBOL);
    repeat PARSERUN;
            if ERRORSET then ERRORRECOVERY;
    until PARSEOVER ;
    1:
end.
```